国外经典教材·计算机科学与技术

Unix/Linux 编程实践教程

（美）Bruce Molay 著
杨宗源 黄海涛 译

清华大学出版社
北 京

目　　录

第1章　Unix系统编程概述 ………… **1**
1.1　介绍 ………… 1
1.2　什么是系统编程 ………… 1
1.2.1　简单的程序模型 ………… 1
1.2.2　系统模型 ………… 2
1.2.3　操作系统的职责 ………… 3
1.2.4　为程序提供服务 ………… 4
1.3　理解系统编程 ………… 4
1.3.1　系统资源 ………… 4
1.3.2　目标：理解系统编程 ………… 5
1.3.3　方法：通过三个问题来理解 ………… 5
1.4　从用户的角度来理解Unix ………… 6
1.4.1　Unix能做些什么 ………… 6
1.4.2　登录—运行程序—注销 ………… 6
1.4.3　目录操作 ………… 8
1.4.4　文件操作 ………… 10
1.5　从系统的角度来看Unix ………… 12
1.5.1　用户和程序之间的连接方式 ………… 12
1.5.2　网络桥牌 ………… 12
1.5.3　bc：Unix的计算器 ………… 13
1.5.4　从bc/dc到Web ………… 16
1.6　动手实践 ………… 16
1.7　工作步骤与概要图 ………… 23
1.7.1　接下来的工作步骤 ………… 23
1.7.2　Unix的概要图 ………… 23
1.7.3　Unix的发展历程 ………… 23
小结 ………… 24

第2章　用户、文件操作与联机帮助：编写who命令 ………… **25**
2.1　介绍 ………… 25
2.2　关于命令who ………… 26
2.3　问题1：who命令能做些什么 ………… 27
2.4　问题2：who命令是如何工作的 ………… 28

2.5　问题 3：如何编写 who …… 32
2.5.1　问题：如何从文件中读取数据结构 …… 33
2.5.2　答案：使用 open、read 和 close …… 34
2.5.3　编写 who1.c …… 36
2.5.4　显示登录信息 …… 37
2.5.5　编写 who2.c …… 39
2.5.6　回顾与展望 …… 44
2.6　编写 cp(读和写) …… 44
2.6.1　问题 1：cp 命令能做些什么 …… 44
2.6.2　问题 2：cp 命令是如何创建/重写文件的 …… 44
2.6.3　问题 3：如何编写 cp …… 45
2.6.4　Unix 编程看起来好像很简单 …… 47
2.7　提高文件 I/O 效率的方法：使用缓冲 …… 48
2.7.1　缓冲区的大小对性能的影响 …… 48
2.7.2　为什么系统调用需要很多时间 …… 48
2.7.3　低效率的 who2.c …… 49
2.7.4　在 who2.c 中运用缓冲技术 …… 50
2.8　内核缓冲技术 …… 53
2.9　文件读写 …… 54
2.9.1　注销过程：做了些什么 …… 54
2.9.2　注销过程：如何工作的 …… 54
2.9.3　改变文件的当前位置 …… 55
2.9.4　编写终端注销的代码 …… 57
2.10　处理系统调用中的错误 …… 58
小结 …… 59

第 3 章　目录与文件属性：编写 ls …… 63
3.1　介绍 …… 63
3.2　问题 1：ls 命令能做什么 …… 63
3.2.1　ls 可以列出文件名和文件的属性 …… 63
3.2.2　列出指定目录或文件的信息 …… 64
3.2.3　经常用到的命令行选项 …… 65
3.2.4　问题 1 的答案 …… 65
3.3　文件树 …… 65
3.4　问题 2：ls 是如何工作的 …… 66
3.4.1　什么是目录 …… 66
3.4.2　是否可以用 open、read 和 close 来操作目录 …… 66
3.4.3　如何读目录的内容 …… 67

3.5　问题 3：如何编写 ls …… 69
3.6　编写 ls -l …… 71
3.6.1　问题 1：ls -l 能做些什么 …… 71
3.6.2　问题 2：ls -l 是如何工作的 …… 72
3.6.3　用 stat 得到文件信息 …… 72
3.6.4　stat 提供的其他信息 …… 74
3.6.5　如何实现 …… 75
3.6.6　将模式字段转换成字符 …… 75
3.6.7　将用户/组 ID 转换成字符串 …… 79
3.6.8　编写 ls2.c …… 81
3.7　三个特殊的位 …… 86
3.7.1　set-user-ID 位 …… 86
3.7.2　set-group-ID 位 …… 87
3.7.3　sticky 位 …… 87
3.7.4　用 ls -l 看到的特殊属性 …… 87
3.8　ls 小结 …… 88
3.9　设置和修改文件的属性 …… 88
3.9.1　文件类型 …… 88
3.9.2　许可位与特殊属性位 …… 89
3.9.3　文件的链接数 …… 90
3.9.4　文件所有者与组 …… 90
3.9.5　文件大小 …… 91
3.9.6　时间 …… 91
3.9.7　文件名 …… 91
小结 …… 92

第 4 章　文件系统：编写 pwd …… 96
4.1　介绍 …… 96
4.2　从用户的角度看文件系统 …… 97
4.2.1　目录和文件 …… 97
4.2.2　目录命令 …… 97
4.2.3　文件操作命令 …… 97
4.2.4　针对目录树的命令 …… 99
4.2.5　目录树的深度几乎没有限制 …… 99
4.2.6　Unix 文件系统小结 …… 100
4.3　Unix 文件系统的内部结构 …… 100
4.3.1　第一层抽象：从磁盘到分区 …… 100
4.3.2　第二层抽象：从磁盘到块序列 …… 100

4.3.3 第三层抽象：从块序列到三个区域的划分 …… 100
4.3.4 文件系统的实现：创建一个文件的过程 …… 101
4.3.5 文件系统的实现：目录的工作过程 …… 102
4.3.6 文件系统的实现：cat 命令的工作原理 …… 104
4.3.7 i-节点和大文件 …… 105
4.3.8 Unix 文件系统的改进 …… 106
4.4 理解目录 …… 107
4.4.1 理解目录结构 …… 107
4.4.2 与目录树相关的命令和系统调用 …… 109
4.5 编写 pwd …… 113
4.5.1 pwd 的工作过程 …… 113
4.5.2 pwd 的一种版本 …… 114
4.6 多个文件系统的组合：由多棵树构成的树 …… 116
4.6.1 装载点 …… 117
4.6.2 多重 i-节点号和设备交叉链接 …… 118
4.6.3 符号链接 …… 119
小结 …… 120

第 5 章 连接控制：学习 stty …… 124
5.1 为设备编程 …… 124
5.2 设备就像文件 …… 124
5.2.1 设备具有文件名 …… 125
5.2.2 设备和系统调用 …… 125
5.2.3 例子：终端就像文件 …… 126
5.2.4 设备文件的属性 …… 126
5.2.5 编写 write 程序 …… 127
5.2.6 设备文件和 i-节点 …… 128
5.3 设备与文件的不同之处 …… 129
5.4 磁盘连接的属性 …… 130
5.4.1 属性 1：缓冲 …… 130
5.4.2 属性 2：自动添加模式 …… 132
5.4.3 用 open 控制文件描述符 …… 133
5.4.4 磁盘连接小结 …… 134
5.5 终端连接的属性 …… 135
5.5.1 终端的 I/O 并不如此简单 …… 135
5.5.2 终端驱动程序 …… 137
5.5.3 stty 命令 …… 137
5.5.4 编写终端驱动程序：关于设置 …… 138

5.5.5　编写终端驱动程序：关于函数 …… 139
5.5.6　编写终端驱动程序：关于位 …… 140
5.5.7　编写终端驱动程序：几个程序例子 …… 142
5.5.8　终端连接小结 …… 146
5.6　其他设备编程：ioctl …… 146
5.7　文件、设备和流 …… 147
小结 …… 147

第6章　为用户编程：终端控制和信号 …… 153
6.1　软件工具与针对特定设备编写的程序 …… 153
6.2　终端驱动程序的模式 …… 154
6.2.1　规范模式：缓冲和编辑 …… 155
6.2.2　非规范处理 …… 156
6.2.3　终端模式小结 …… 157
6.3　编写一个用户程序：play_again.c …… 158
6.4　信号 …… 168
6.4.1　Ctrl-C做什么 …… 168
6.4.2　信号是什么 …… 168
6.4.3　进程该如何处理信号 …… 170
6.4.4　信号处理的例子 …… 171
6.5　为处理信号做准备：play_again4.c …… 173
6.6　进程终止 …… 176
6.7　为设备编程 …… 176
小结 …… 176

第7章　事件驱动编程：编写一个视频游戏 …… 180
7.1　视频游戏和操作系统 …… 180
7.2　任务：单人弹球游戏(Pong) …… 182
7.3　屏幕编程：curses库 …… 182
7.3.1　介绍curses …… 182
7.3.2　curses内部：虚拟和实际屏幕 …… 185
7.4　时间编程：sleep …… 185
7.5　时钟编程1：Alarms …… 188
7.5.1　添加时延：sleep …… 189
7.5.2　sleep()是如何工作的：使用Unix中的Alarms …… 189
7.5.3　调度将要发生的动作 …… 191
7.6　时间编程2：间隔计时器 …… 191
7.6.1　添加精度更高的时延：usleep …… 192
7.6.2　三种计时器：真实、进程和实用 …… 192

7.6.3　两种间隔：初始和重复 …… 192
7.6.4　用间隔计时器编程 …… 193
7.6.5　计算机有几个时钟 …… 196
7.6.6　计时器小结 …… 197
7.7　信号处理 1：使用 signal …… 198
7.7.1　早期的信号处理机制 …… 198
7.7.2　处理多个信号 …… 198
7.7.3　测试多个信号 …… 200
7.7.4　信号机制其他的弱点 …… 202
7.8　信号处理 2：sigaction …… 203
7.8.1　处理多个信号：sigaction …… 203
7.8.2　信号小结 …… 206
7.9　防止数据损毁(Data Corruption) …… 206
7.9.1　数据损毁的例子 …… 206
7.9.2　临界区(Critical Sections) …… 206
7.9.3　阻塞信号：sigprocmask 和 sigsetops …… 207
7.9.4　重入代码(Reentrant Code)：递归调用的危险 …… 208
7.9.5　视频游戏中的临界区 …… 208
7.10　kill：从另一个进程发送的信号 …… 209
7.11　使用计时器和信号：视频游戏 …… 210
7.11.1　bounce1d.c：在一条线上控制动画 …… 210
7.11.2　bounce2d.c：两维动画 …… 213
7.11.3　完成游戏 …… 218
7.12　输入信号：异步 I/O …… 218
7.12.1　使用异步 I/O …… 218
7.12.2　方法 1：使用 O_ASYNC …… 218
7.12.3　方法 2：使用 aio_read …… 221
7.12.4　弹球程序中需要异步读入吗 …… 224
7.12.5　异步输入、视频游戏和操作系统 …… 224
小结 …… 224

第 8 章　进程和程序：编写命令解释器 sh …… 228
8.1　进程=运行中的程序 …… 228
8.2　通过命令 ps 学习进程 …… 229
8.2.1　系统进程 …… 231
8.2.2　进程管理和文件管理 …… 232
8.2.3　内存和程序 …… 232
8.3　shell：进程控制和程序控制的一个工具 …… 233

8.4 shell 是如何运行程序的 ······ 234
8.4.1 shell 的主循环 ······ 234
8.4.2 问题 1：一个程序如何运行另一个程序 ······ 235
8.4.3 问题 2：如何建立新的进程 ······ 240
8.4.4 问题 3：父进程如何等待子进程的退出 ······ 244
8.4.5 小结：shell 如何运行程序 ······ 249
8.5 实现一个 shell：psh2.c ······ 250
8.6 思考：用进程编程 ······ 254
8.7 exit 和 exec 的其他细节 ······ 255
8.7.1 进程死亡：exit 和_exit ······ 255
8.7.2 exec 家族 ······ 256
小结 ······ 257

第 9 章 可编程的 shell、shell 变量和环境：编写自己的 shell ······ 260
9.1 shell 编程 ······ 260
9.2 什么是以及为什么要使用 shell 脚本语言 ······ 260
9.3 smsh1——命令行解析 ······ 263
9.4 shell 中的流程控制 ······ 270
9.4.1 if 语句做些什么 ······ 270
9.4.2 if 是如何工作的 ······ 271
9.4.3 在 smsh 中增加 if ······ 272
9.4.4 smsh2.c：修改后的代码 ······ 273
9.5 shell 变量：局部和全局 ······ 278
9.5.1 使用 shell 变量 ······ 279
9.5.2 变量的存储 ······ 280
9.5.3 增加变量命令：Built-ins ······ 280
9.5.4 效果如何 ······ 283
9.6 环境：个性化设置 ······ 284
9.6.1 使用环境 ······ 285
9.6.2 什么是环境以及它是如何工作的 ······ 286
9.6.3 在 smsh 中增加环境处理 ······ 288
9.6.4 varlib.c 的代码 ······ 290
9.7 已实现的 shell 的功能 ······ 295
小结 ······ 296

第 10 章 I/O 重定向和管道 ······ 299
10.1 shell 编程 ······ 299
10.2 一个 shell 应用程序：监视系统用户 ······ 300
10.3 标准 I/O 与重定向的若干概念 ······ 301

10.3.1 概念 1：3 个标准文件描述符 …… 302
10.3.2 默认的连接：tty …… 302
10.3.3 程序都输出到 stdout …… 303
10.3.4 重定向 I/O 的是 shell 而不是程序 …… 303
10.3.5 理解 I/O 重定向 …… 304
10.3.6 概念 2："最低可用文件描述符(Lowest-Available-fd)"原则 …… 304
10.3.7 两个概念的结合 …… 305
10.4 如何将 stdin 定向到文件 …… 305
10.4.1 方法 1：close then open …… 305
10.4.2 方法 2：open..close..dup..close …… 308
10.4.3 系统调用 dup 小结 …… 310
10.4.4 方法 3：open..dup2..close …… 310
10.4.5 shell 为其他程序重定向 stdin …… 310
10.5 为其他程序重定向 I/O：who> userlist …… 310
10.6 管道编程 …… 314
10.6.1 创建管道 …… 314
10.6.2 使用 fork 来共享管道 …… 317
10.6.3 使用 pipe、fork 以及 exec …… 318
10.6.4 技术细节：管道并非文件 …… 320
小结 …… 321

第 11 章 连接到近端或远端的进程：服务器与 Socket(套接字) …… 325
11.1 产品和服务 …… 325
11.2 一个简单的比喻：饮料机接口 …… 326
11.3 bc：Unix 中使用的计算器 …… 327
11.3.1 编写 bc：pipe、fork、dup、exec …… 328
11.3.2 对协同进程的讨论 …… 332
11.3.3 fdopen：让文件描述符像文件一样使用 …… 332
11.4 popen：让进程看似文件 …… 332
11.4.1 popen 的功能 …… 332
11.4.2 实现 popen：使用 fdopen 命令 …… 334
11.4.3 访问数据：文件、应用程序接口(API)和服务器 …… 336
11.5 socket：与远端进程相连 …… 337
11.5.1 类比："电话中传来声音：现在时间是……" …… 338
11.5.2 因特网时间、DAP 和天气服务器 …… 340
11.5.3 服务列表：众所周知的端口 …… 341
11.5.4 编写 timeserv.c：时间服务器 …… 342

11.5.5　测试 timeserv.c …… 347
11.5.6　编写 timeclnt.c：时间服务客户端 …… 347
11.5.7　测试 timeclnt.c …… 350
11.5.8　另一种服务器：远程的 ls …… 351
11.6　软件精灵 …… 356
小结 …… 356

第 12 章　连接和协议：编写 Web 服务器 …… 360
12.1　服务器设计重点 …… 360
12.2　三个主要操作 …… 360
12.3　操作 1 和操作 2：建立连接 …… 361
12.3.1　操作 1：建立服务器端 socket …… 361
12.3.2　操作 2：建立到服务器的连接 …… 362
12.3.3　socklib.c …… 362
12.4　操作 3：客户/服务器的会话 …… 364
12.4.1　使用 socklib.c 的 timeserv/timeclnt …… 365
12.4.2　第 2 版的服务器：使用 fork …… 366
12.4.3　服务器的设计问题：DIY 或代理 …… 367
12.5　编写 Web 服务器 …… 369
12.5.1　Web 服务器功能 …… 369
12.5.2　设计 Web 服务器 …… 370
12.5.3　Web 服务器协议 …… 370
12.5.4　编写 Web 服务器 …… 372
12.5.5　运行 Web 服务器 …… 374
12.5.6　Webserv 的源程序 …… 374
12.5.7　比较 Web 服务器 …… 379
小结 …… 379

第 13 章　基于数据报(Datagram)的编程：编写许可证服务器[①] …… 381
13.1　软件控制 …… 381
13.2　许可证控制简史 …… 382
13.3　一个非计算机系统实例：轿车管理系统 …… 383
13.3.1　轿车钥匙管理描述 …… 383
13.3.2　用客户/服务器方式管理轿车 …… 383
13.4　许可证管理 …… 384
13.4.1　许可证服务系统：它做些什么 …… 384
13.4.2　许可证服务系统：如何工作 …… 385
13.4.3　一个通信系统的例子 …… 386
13.5　数据报 socket …… 386

13.5.1 流与数据报的比较 …… 387
13.5.2 数据报编程 …… 388
13.5.3 sendto 和 recvfrom 的小结 …… 393
13.5.4 数据报应答 …… 394
13.5.5 数据报小结 …… 396
13.6 许可证服务器版本 1.0 …… 396
13.6.1 客户端版本 1 …… 397
13.6.2 服务器端版本 1 …… 401
13.6.3 测试版本 1 …… 406
13.6.4 进一步的工作 …… 407
13.7 处理现实的问题 …… 407
13.7.1 处理客户端崩溃 …… 407
13.7.2 处理服务器崩溃 …… 410
13.7.3 测试版本 2 …… 412
13.8 分布式许可证服务器 …… 414
13.9 Unix 域 socket …… 416
13.9.1 文件名作为 socket 地址 …… 416
13.9.2 使用 Unix 域 socket 编程 …… 416
13.10 小结：socket 和服务器 …… 419
小结 …… 419

第 14 章 线程机制：并发函数的使用 …… 423
14.1 同一时刻完成多项任务 …… 423
14.2 函数的执行路线 …… 424
14.2.1 一个单线程程序 …… 424
14.2.2 一个多线程程序 …… 425
14.2.3 相关函数小结 …… 427
14.3 线程间的分工合作 …… 428
14.3.1 例 1：incrprint.c …… 428
14.3.2 例 2：twordcount.c …… 430
14.3.3 线程内部的分工合作：小结 …… 436
14.4 线程与进程 …… 437
14.5 线程间互通消息 …… 438
14.5.1 通知选举中心 …… 439
14.5.2 使用条件变量编写程序 …… 440
14.5.3 使用条件变量的函数 …… 443
14.5.4 回到 Web 服务器的例子 …… 444
14.6 多线程的 Web 服务器 …… 444

14.6.1 Web 服务器程序的改进 …… 445
14.6.2 多线程版本允许一个新的功能 …… 445
14.6.3 防止僵尸线程(Zombie Threads)：独立线程 …… 445
14.6.4 Web 服务器代码 …… 445
14.7 线程和动画 …… 452
14.7.1 使用线程的优点 …… 452
14.7.2 多线程版本的 bounce1d.c …… 453
14.7.3 基于多线程机制的多重动画：tanimate.c …… 455
14.7.4 tanimate.c 中的互斥量 …… 458
14.7.5 屏幕控制线程 …… 459
小结 …… 460

第 15 章 进程间通信(IPC) …… 464
15.1 编程方式的选择 …… 464
15.2 talk 命令：从多个数据源读取数据 …… 465
15.2.1 同时从两个文件描述符读取数据 …… 465
15.2.2 select 系统调用 …… 466
15.2.3 select 与 talk …… 469
15.2.4 select 与 poll …… 469
15.3 通信的选择 …… 469
15.3.1 一个问题的三种解决方案 …… 470
15.3.2 通过文件的进程间通信 …… 470
15.3.3 命名管道 …… 471
15.3.4 共享内存 …… 473
15.3.5 各种进程间通信方法的比较 …… 475
15.4 进程之间的分工合作 …… 476
15.4.1 文件锁 …… 476
15.4.2 信号量(Semaphores) …… 480
15.4.3 socket 及 FIFO 与共享的存储 …… 487
15.5 打印池 …… 488
15.5.1 多个写者、一个读者 …… 488
15.5.2 客户/服务器模型 …… 489
15.6 纵观 IPC …… 490
15.7 连接与游戏 …… 492
小结 …… 493

第 1 章　Unix 系统编程概述

概念

- Unix 系统包含用户程序和系统内核
- 内核由多个子系统构成
- 内核管理所有的程序和资源
- 进程之间的通信对 Unix 程序是很重要的
- 什么是系统编程

相关命令

- bc
- more

1.1 介　　绍

什么是系统编程？什么是 Unix 系统编程？本书具体会涉及哪些知识？本章力图回答上述问题。

首先从分析操作系统的职责入手，来解释如何编写与操作系统紧密相关的程序。然后通过分析标准的 Unix 命令，以及它们用到的系统调用，进一步指导读者自己编程实现相应的功能。这一章的最后会通过一幅图来描述 Unix 系统。本书的主要学习形式就是通过图示和剖析文中程序所涉及的命令、技术，进而实现系统编程。

1.2 什么是系统编程

1.2.1 简单的程序模型

你可能写过各种各样的程序，有科学计算方面的、金融方面的、图像方面的、文字处理方面的等。大部分的程序都是基于以下模型，如图 1.1 所示。

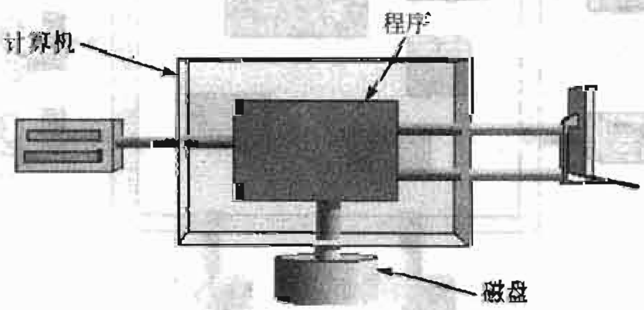


图 1.1　计算机中的程序

在这个模型中，程序就是可以在计算机上运行的一段代码，程序把输入数据做相应处理后输出。例如用户在键盘上输入数据，然后在屏幕得到输出。程序可能对磁盘进行操作，还可能会用到打印机。

遵循上述模型，看以下代码：

```
/* copy from stdin to stdout */
main()
{
    int c;
    while( ( c = getchar() ) != EOF )
        putchar(c);
}
```

这段代码对应图 1.2 所示的模型。

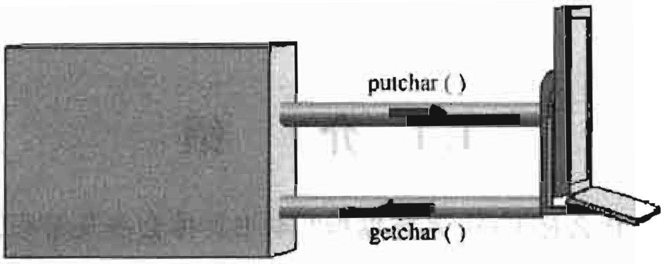


图 1.2　程序的输入 l 输出

在图 1.2 中键盘和显示器与程序直接相连。在简单的个人计算机中，实际情况是很类似的，键盘和显示卡直接连到计算机的主板上，CPU 和内存也是通过插槽直接连在主板上，它们通过主板上的印刷线路，连为一体。如果能够打开机箱，所看到的大致如此。

1.2.2　系统模型

如果所使用的系统是一个多用户系统，如典型的 Unix 系统，那会是一副怎样的情形呢？

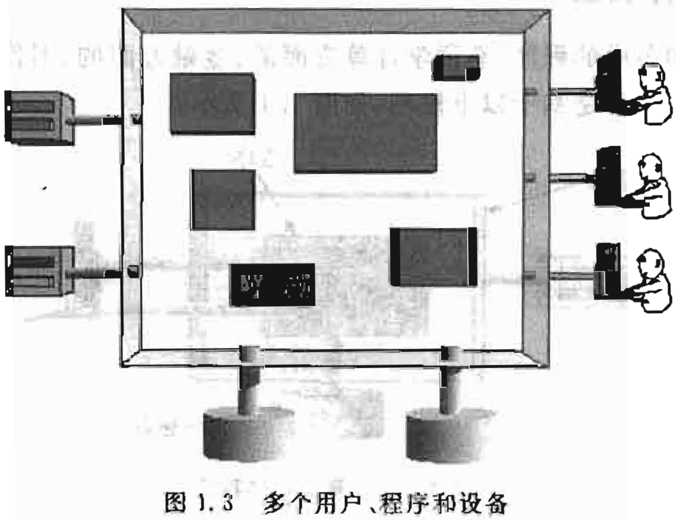


图 1.3　多个用户、程序和设备

刚才的简单模型已经不适用,图 1.3 会更接近一些。

在这个系统中有多个用户同时运行多个程序,可能需要访问多个设备。

虽然模型复杂了,但对程序而言,它还是从键盘得到数据,将结果显示在显示器上,也可以对磁盘读写,这些操作都没有任何问题,它使用的还是简单模型。

接下来考虑一种更为复杂的情况,有许多键盘/显示器,它们可以随意地连接到不同的程序,随意地操作它们,这种情况如图 1.4 所示。

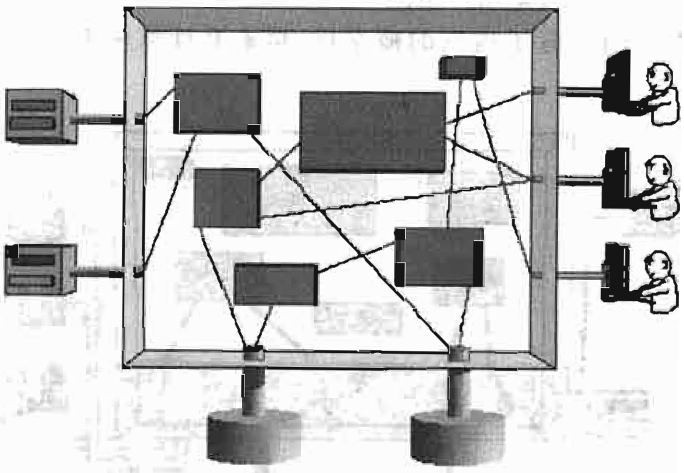


图 1.4 终端可以随意地连接到程序

实际上,在计算机内部,这种随意的连接是不允许的,必须采用一种机制进行管理。

1.2.3 操作系统的职责

计算机用操作系统来管理所有的资源,并将不同的设备和不同的程序连接起来。从连接的角度来讲,操作系统的作用就像主板上的印刷线路一样。

有了操作系统以后,图 1.4 的混乱状态就可以得到改变,新的模型如图 1.5 所示。

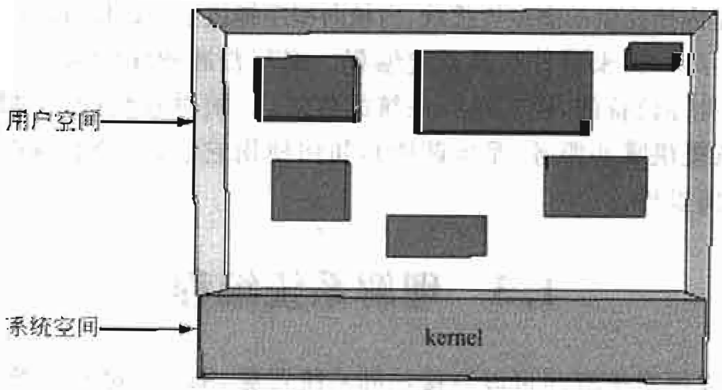


图 1.5 操作系统是一个特殊的程序

操作系统也是程序,与普通程序一样,也运行在内存中,同时它又是一个特殊的程序,它能把普通程序与其他程序或设备连接起来。

1.2.4 为程序提供服务

现在的问题(系统中的多个用户和程序是如何连接起来的)和大致的解决办法(通过一个管理程序)已经很清楚了,接下来看具体的解决方案。

首先要解释一些术语,内存空间用来存放程序和数据,就像古雅典人腾出空间来放衣服一样,所有的程序都必须在内存空间中才能运行,用来容纳操作系统的内存空间叫做系统空间,容纳应用程序的内存空间叫做用户空间。

操作系统也被称为内核,有了内核的概念后,再来看计算机系统的连接情况,如图 1.6 所示。

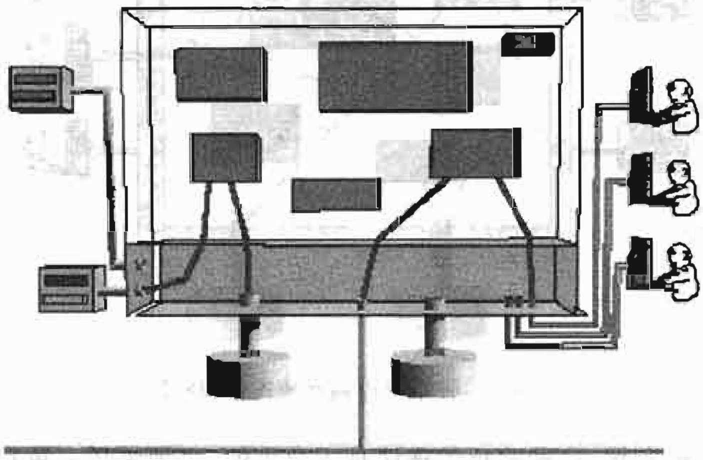


图 1.6 内核管理计算机系统的连接

注意,在图 1.6 中可以发现,程序要访问设备(如键盘、磁盘和打印机)必须通过内核,所以只有内核才能直接管理设备。

程序如果要从键盘得到数据,必须向内核发出请求,若在显示器上显示结果,也要通过内核,程序中所有对设备的操作都是通过内核进行的。

图 1.6 中的线是内核提供的虚拟连接线,内核向程序提供服务以便程序能够访问到设备。

解释了这些内容后,再来看什么是系统编程。编写普通程序时可以认为,程序是直接连到键盘、显示器、磁盘等设备的,但在进行系统编程时,必须对系统的结构和工作方式有更深的了解,要知道内核提供哪些服务(系统调用),如何使用它们,系统有哪些资源和设备,不同的资源和设备该如何操作。

1.3 理解系统编程

内核提供服务以便系统程序可以直接访问系统资源,那么有哪些系统资源和服务呢?

1.3.1 系统资源

1、处理器(Processor)

程序是由指令构成的,处理器是执行指令的硬件设备,一个系统中可能有多个处理器。

内核能够安排一个程序何时开始执行，何时暂时停止、恢复执行，何时终止执行。

2. 输入输出(I/O)

程序中所有输入/输出的数据、终端的输入/输出数据还有硬盘输入/输出数据，都必须流经内核，这种集中的处理方式有以下优点：正确性，数据流不会流错地方；有效性，程序员无需考虑不同设备之间的差异；安全性，数据信息不会被未被授权的程序非法访问。

3. 进程管理(Process Management)

进程指程序的一次运行，每个进程都有自己的资源，如内存、打开的文件和其他运行时所需的系统资源。内核中与进程相关的服务有新建一个进程、中止进程、进程调度等。

4. 内存(Memory)

内存是计算机系统中很重要的资源，程序必须被装载到内存中才可以运行。内核的职责之一是内存管理，在需要的时候给程序分配内存，当程序不需要的时候回收内存，内核还能够保证内存不被其他的进程非法访问。

5. 设备(Device)

计算机系统中可以有各种各样的外设，如磁带机、光驱、鼠标、扫描仪和数码摄像机等，它们的操作方式各不相同，内核能屏蔽掉这种差异，使得对设备的操作方式简单而统一。例如，一个程序想要从数码照相机中取出照片存储在计算机中，它只需向内核提出操作该资源的请求即可。

6. 计时器(Timers)

程序的工作与时间有关，有的需要定时被触发，有的需要等一段时间再开始某个动作，有的需要知道某一个操作消耗的时间，这些都涉及计时器，内核可以通过系统调用向应用程序提供计时器服务。

7. 进程间通信(Interprocess Communication)

在现实生活中人们通过电话、e-mail、信件、广播、电视等互相通信，在计算机的世界中，不同的进程也需要互相通信，内核提供的服务使进程间通信成为可能。就像电信和邮政提供的服务，通信也是资源。

8. 网络(Networking)

网络之间的通信可以看作是进程间通信的特殊形式，通过网络，不同主机上的进程，即使使用的是不同操作系统，也可以互相通信。网络通信也是内核提供的服务。

1.3.2 目标：理解系统编程

刚才已经简单介绍了内核所提供的各种类型的服务，各种内核服务具体有什么特点，会用到哪些设备，需要哪些参数，会提供哪些数据，接下来的目标是掌握内核服务的机制，以便在自己的程序中使用这些服务。

1.3.3 方法：通过三个问题来理解

本书通过以下 3 个步骤来学习。

1. 分析程序

首先分析现有的程序，了解它的功能及实现原理。

2. 学习系统调用

看程序都用到哪些系统调用，以及每个系统调用的功能和使用方法。

3. 编程实现

利用学到的原理和系统调用，自己编程实现原来程序所实现的功能。

以上 3 步可以通过下面 3 个问题来实现：

- 它能做什么？
- 它是如何实现的？
- 能不能自己编写一个？

1.4　从用户的角度来理解 Unix

1.4.1　Unix 能做些什么

要学习 Unix 系统编程，首先看看 Unix 能做些什么。下面从一个从终端登录到系统中的普通用户的角度来看 Unix 是什么，它能做些什么。

1.4.2　登录—运行程序—注销

使用 Unix 的过程一般如下：登录到系统中，运行程序，工作结束后再注销。登录的时候要输入用户名和密码：

```
Linux 1.2.13 (maya) (ttyp1)
maya login: betsy
Password: _
```

登录到系统中以后，就可以运行各种各样的程序了，比如收发邮件程序、科学计算程序以及游戏程序。

运行程序也很简单，当显示器上出现提示符后（一般是“$”），输入程序的名字，按回车，程序将开始运行。运行结束后，提示符再次出现。

图形用户界面的操作方式实际上也是这样的。图标和菜单可以看作是提示符，双击图标就像运行命令一样，系统会把双击操作解释为相应程序的执行。

运行完程序后，可以从系统中注销：

```
$ exit
```

在有些系统中，可以通过输入 logout 或按组合键 Ctrl+D 来注销。

它是如何工作的呢？

这看起来很简单，但系统在内部是如何处理的呢？

首先来看登录过程。人们常说使用计算机就像使用自己的汽车一样，这是个人计算机中的概念，问题在于 Unix 允许许多用户，可能是几十甚至几百人，同时登录到系统中，系统

是如何对这么多的用户进行管理的呢？

在登录过程中，当用户名和密码通过验证后，系统会启动一个叫 shell 的进程，然后把用户交给这个进程，由这个进程处理用户的请求。每个用户都有属于自己的 shell 进程。

图 1.7 是用户登录到 Unix 系统中的示意图。

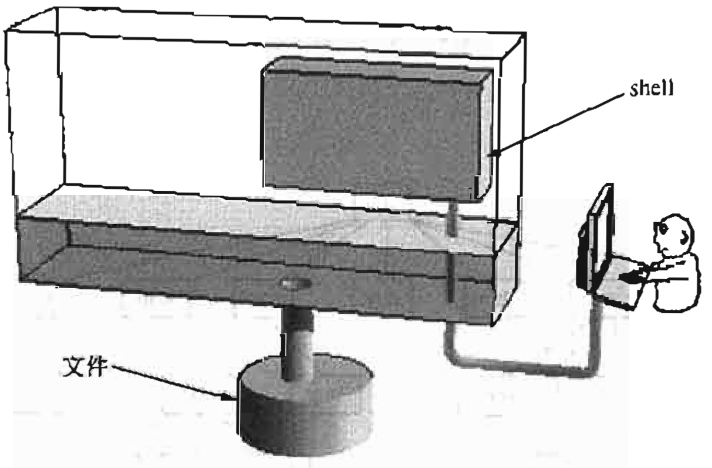


图 1.7 用户登录到系统

图 1.7 中左边的大盒子表示计算机系统，坐在键盘和显示器前的是用户，计算机里有内存，内核运行在内存中，shell 为用户提供服务，shell 和用户之间的连接由内核控制。

Shell 在屏幕上显示出提示符，表示现在可以接收用户的输入。对于普通用户而言提示符一般是“$”，也可能还会显示其他的提示信息。用户可以在提示符后输入要运行的程序的名字，内核负责把用户的输入传给 shell。例如，运行显示日期和时间的程序如下：

```
$
$ date
Sat Jul 1 21:34:10 EDT 2000
$ _
```

date 命令显示出日期，接下来显示命令提示符。

要运行其他命令，只要输入程序名即可，Unix 中有一个程序 fortune，下面是它运行的例子：

```
$ fortune
Algol - 60 surely must be regarded as the most important
programming language yet developed.
    -- T. Cheatham
$ _
```

当用户注销时，内核会结束所有分配给这个用户的进程。

内核是如何创建 shell 进程的呢？shell 进程是如何得到输入的程序名，内核又是如何运行程序的呢？登录系统和运行程序并非想像的那么简单。这些细节会在第 8 章讨论。

1.4.3　目录操作

用户登录后，可以对自己的文件进行操作。文件中可以有 e-mail、图片、源程序、可执行程序等各种各样的数据。文件被组织在目录中。

1. 目录树

在 Unix 系统中，文件和目录被组织成树状结构，Unix 提供相应的命令来对目录进行操作。

图 1.8 是一个目录树的例子。

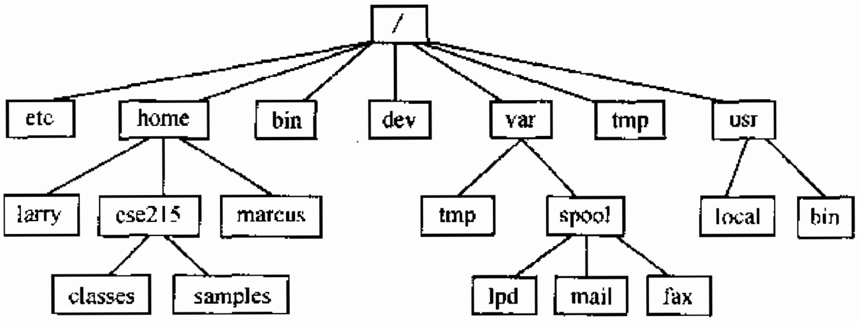


图 1.8　目录树的一部分

如图 1.8 所示，文件系统的最顶端是"/"，它叫做根目录，根目录一般都包含几个子目录。大多数的 Unix 系统都在根目录下面有/etc、/home、/bin 等几个子目录，它们都有特定的用途，比如大多数的 Unix 用户都有自己的主目录，而一般来说用户主目录就在/home 目录中。

Unix 系统中有很多命令都与目录有关，如新建目录、删除目录、改变当前目录、列出目录内容等，读完本节的内容后可以自己试一试这些命令。

2. 目录操作命令

(1) ls——列出目录内容

ls 命令的作用是列出目录的内容，即当前目录里的文件和子目录，如果只输入 ls，那么列出的是当前目录的内容，如果输入 ls dirname，那么列出的是 dirname 所指定的目录的内容，如输入：

```
ls /etc
```

会列出/etc 目录里面所包含的文件和子目录，同样地，如果输入：

```
ls /
```

则会列出根目录的内容。

(2) cd——改变当前目录

cd 命令的作用是改变当前的目录。当刚刚登录到系统中时，当前目录是自己的主目录，可以通过 cd 命令转到其他目录，如：

```
cd /bin
```

会转到/bin 目录下，这个目录中有很多系统命令，可以用 ls 查看有哪些命令。通过下述命令转到上一层目录：

```
cd ..
```

无论当前目录是什么，通过下述命令都可以立即回到用户的主目录：

```
cd
```

(3) pwd——显示当前目录

pwd 命令告诉当前目录是什么，它列出的是全路径，即从根目录开始的路径，如：

```
$ pwd
/home/cse215/samples
```

从上述操作可以知道，当前目录是通过从根目录开始，依次进入 home、cse215、samples 来得到。

(4) mkdir、rmdir——新建、删除目录

用 mkdir 来新建目录，如：

```
$ cd
$ mkdir jokes
```

先进入主目录，然后在主目录中建立 jokes 这样一个目录。一般来说，只能在自己的目录中新建目录而不允许在其他用户的目录中新建目录。

要删除一个已经存在的目录，可以用 rmdir 命令，如：

```
$ rmdir jokes
```

如果 jokes 目录是空目录，那这个命令可以把 jokes 删除。注意用 rmdir 来删除目录，必须先把目录中的文件和子目录删除或移走。

3. 目录操作命令的工作原理

从刚才的分析可以知道硬盘上的目录和文件构成了一棵目录树，树的中间结点是目录，每个目录又可以包含很多的子目录和文件，可以新建或删除目录，可以从一个目录转到其他的目录。

然而这些是如何工作的呢？首先来考察硬盘，硬盘是由很多片金属或玻璃的盘片组合起来构成的，这些盘片上可以保存磁性信息，问题是目录在哪里？用户在自己的主目录中意味着什么？转到其他目录又意味着什么？Unix 允许很多用户同时登录到系统中，他们可以有相同的当前目录，也可以在不同的目录中，会不会因为很多用户在同一个目录中导致这个目录过分拥挤？还有的问题是，如果要自己来编写一个改变当前目录的程序，该如何来实现？内核在这棵目录树中扮演什么角色？

1.4.4 文件操作

文件存放在目录中，用户文件位于用户自己的主目录中，系统文件位于系统目录中。对文件的操作有哪些呢？

1. 文件操作的命令

(1) 文件命名规则

每个文件都有文件名，在大多数的 Unix 系统中，文件名最长可以是 250 个字符，很多字符都可以出现在文件名中，如大小写字符、标点符号、空格、tab，甚至回车符，但是不能包含根目录符号“/”。

(2) cat，more，less，pg——查看文件的内容

上述命令都可以用来查看文件的内容，但也有些细微的差别，cat 一下子列出文件的所有内容：

```
$ cat shopping - list
soap
cornflakes
milk
apples
jam
$
```

当文件的内容比较多，在一屏内显示不完时，more 会更加合适：

```
$ more longfile
```

显示一屏后会暂停输出，这时用户按空格键，more 会继续输出下一屏，如果按回车，则会显示下一行，输入“q”则退出。另外两个命令 less 和 pg 的功能与 more 是类似的。

(3) cp——文件复制

可以用 cp 命令来复制文件，如：

```
$ cp shopping - list last.week.list
```

将文件 shopping - list 复制一份，新的文件名为 last.week.list。

(4) rm——文件删除

删除文件的例子如下：

```
$ rm old.data junk shopping.june1992
```

一次删掉了 3 个文件。

Unix 并不提供恢复被删除文件的功能，其中一个原因是 Unix 是一个多用户系统，当一个文件被删掉以后，它所占用的存储空间可能被立即分配给其他用户的文件，有可能某块磁盘空间刚才还是你的学期论文，下一个时刻就变成了另外一个用户的 C 程序，所以成功恢复

的可能性很低。

(5) mv——重命名或移动文件

mv 命令可以更改文件名或动文件，如：

```
$ mv prog1.c first_program.c
```

将文件 prog1.c 改名，新的名字是 first_program.c。

也可以移动文件，即改变文件的位置，如：

```
$ mkdir mycode
$ mv first_program.c mycode
```

新建一个目录 mycode，然后将 first_program.c 移动到这个目录中。

(6) lpr. lp——打印文件

用 lpr 来打印文件，如：

```
$ lpr filename
```

上述命令把 filename 送到默认的打印机打印，很多大型系统有不止一台的打印机，可以通过 lpr 命令指定用哪一台打印，在有些系统中，可用 lp 命令来完成 lpr 的工作。

2. 文件操作命令的工作原理

从用户的角度来看，文件是数据的集合，文件中的数据是如何存储在磁盘上的？文件是如何被复制的？如何移动和改名的？进一步来讲，文件的名字存放在哪里？作为一个系统程序员，必须能够回答这些问题，而这些问题的答案就在 Unix 系统中。

3. 文件许可权限

系统中每个用户都有自己的文件，出于很多原因，你不会希望别的用户能够修改自己的文件，甚至读也不行。系统中有一些管理命令，如果使用得不好会对系统造成损害，管理员不希望普通用户也有运行这些命令的权力。这些对文件和命令操作的限制是如何行使的呢？

Unix 通过一些文件属性来对文件和命令的操作进行控制。每个文件都有文件所有者(owner)和文件许可权限。文件所有者指明了系统中某一个用户，文件的创建者就是文件所有者。

文件许可权限分为 3 组，通过 ls -l 命令可以看到：

```
$ ls -l outline.01
-rwxr-x---   1   molay   users   1064   Jun   29 00:39   outline.01
```

-l 称为命令行选项，选项 -l 使得 ls 输出文件的详细信息。同一个命令，可以通过不同的选项，使它所做的工作稍有不同。这里的文件详细信息包含文件的许可权限、文件所有者、文件长度、最后修改时间等，其中的“rwxr-x---”就是文件许可权限。

每个文件都有文件所有者和 3 组许可权限：

```
- r w x    r w x    r w x    r:read,    w:write,    x:execute
  user     group    other
```

与 3 组许可权限相对应,用户也被分为 3 组: user,文件所有者; group,与文件所有者同组的用户; other,其他用户。每组的用户都可以有 3 种权限: 读权限、写权限和执行权限。这样针对不同用户一共是 9 个权限,这些权限可以分别设定,如可以指定其他用户只能修改文件而不能读文件,文件所有者甚至可以取消自己读自己文件的权限。

4. 文件许可权限的工作原理

文件许可权限是如何工作的? 怎么来设置? 系统是如何应用刚才讲到的权限的? 许可权限存放在哪里? 在后续的章节会对这些问题做解答。

1.5 从系统的角度来看 Unix

1.5.1 用户和程序之间的连接方式

前面的小节从用户的角度来看 Unix 系统,用户登录到系统中,运行程序,再从系统中退出。与此同时,可能有很多其他用户也在登录、运行程序或退出系统,他们可以同时对一个文件进行操作,好像工作在各自独立的空间中,他们之间还可以通过发送 e-mail 或即时消息进行沟通。

其实每个独立的空间都是系统的一部分,从宏观的角度来看,系统还可能由很多的用户、很多程序,甚至很多计算机系统互相连接而成。接下来,将通过 3 个例子来看系统是如何工作的,如何编程使系统中的不同部分做到协调统一。

1.5.2 网络桥牌

许多人都玩网络桥牌这个游戏,世界各地的玩家通过计算机网络连接到一起,游戏开始以后,参加者就可以看到一个共同的牌桌,能够看到别人出的牌,图 1.9 是一个简单的说明。

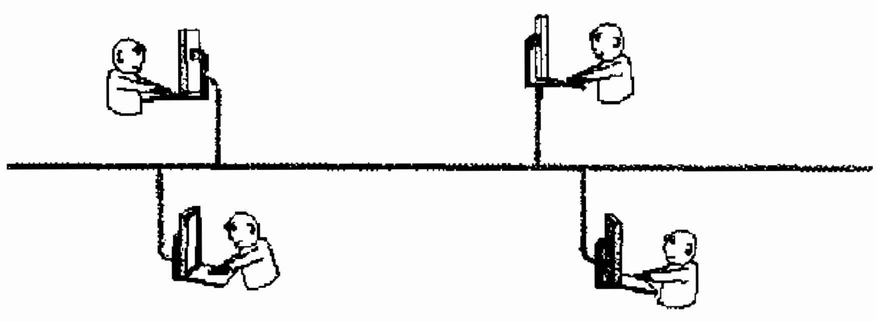


图 1.9　4 个人通过网络打桥牌

图 1.9 中有 4 个人,他们每人都有一台计算机,通过网络连接在一起。但图 1.9 中还少了牌桌,下面把它加上,如图 1.10 所示。

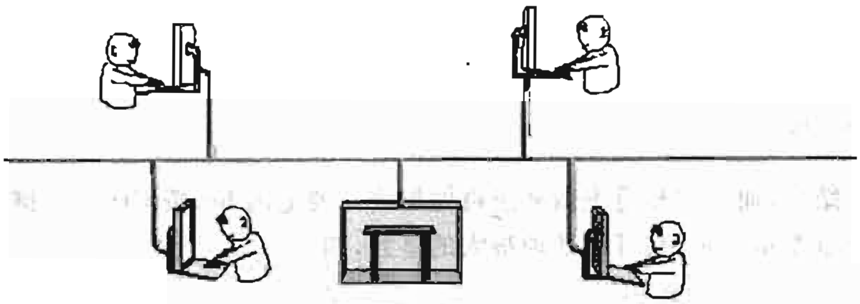


图 1.10　服务器上的牌桌

图 1.10 中增加了第 5 个实体——牌桌，牌桌位于服务器上，在 4 个玩家的眼里，牌是放在牌桌上的，他们也是通过牌桌才能够开始游戏。

在现实生活中玩牌的时候，人们轮流出牌，但在网络游戏中，是由谁来控制该哪一个人出牌？牌又存放在哪里？某个人手中有几张牌又意味着什么？如何来保证不让两个人有同一张牌？这在现实生活中不会有任何问题，但在虚拟的网络中确实要仔细考虑。

图 1.11 显示了在网络桥牌中的信息流。

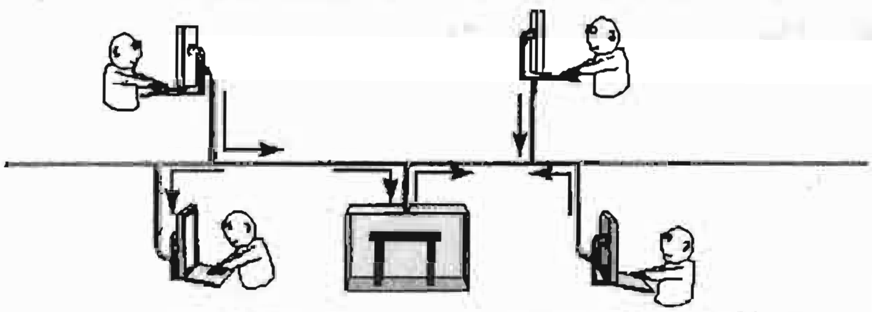


图 1.11　分布的程序向其他用户发送信息

网络桥牌的例子展示了 Unix 系统编程中 3 个重要的方面。

(1) 通信

某个用户或进程如何与其他用户或进程交换信息？

(2) 协作

在同一个时刻，网络桥牌的两个用户不会都去拿同一张牌，程序如何来协调多个进程使他们能够没有冲突地访问共享资源？

(3) 网络访问

在这个例子中，互相独立的计算机通过网络连接到一起，那么计算机中的程序是如何来使用网络的呢？

1.5.3　bc：Unix 的计算器

Unix 系统中的 bc 命令是执行一个基于字符的计算器程序，bc 有两个重要的特点，稍后会讲到。要启动这个计算器，只要输入：

```
$ bc
```

不会有任何的版本信息和提示信息出现，但这时已经可以接收输入了，如：

```
2 + 3 * 4 + 5 * 10
```

bc 会显示计算结果，而且它知道先做乘法再做加法。要退出 bc，按 Ctrl+D 键。

bc 的一个重要特点是，它可以处理很大的整数，如：

```
99999999999999999999 * 88888888888888888888
8888888888888888888871111111111111111112
```

为了得到更大的整数，可以借助于幂运算：

```
3333 ^ 44
1011006158449564099500589848918228579482240528849807070336511\
1794769438904110649252911543814688907219481422090046883818703\
5540915541156321805747562427309521
```

3333 的 44 次方，得到的整数足足有两行半那么长，bc 还是可编程的，可以定义变量，有逻辑判断和循环结构，语法与 C 语言类似，如：

```
x = 3
if ( x == 3 ){
   y = x * 3;
}
y
```

bc 的另一个重要特点是，从严格的意义上讲，bc 并不做任何计算。为了说明这一点，做如下操作：

```
$ bc
2 + 3
5
<-- press Ctrl - Z here
Stopped
$ ps
PID    TTY    S   TIME      CMD
25102  ttyp2  T   0:00.02   bc
27081  ttyp2  T   0:00.01   dc -
27560  ttyp2  I   0:00.59   - bash
27681  ttyp2  T   0:00.00   bc
$ fg
<-- press Ctrl - D here
```

ps 命令可以列出系统中运行的所有进程，这里一共有 4 个进程，除了两个 bc 外，bash 是

shell进程，那么dc是什么？

在大多数的Unix系统中都提供了联机帮助，可以从那里得到需要的信息，要找关于dc的信息，只要输入：

```
$ man dc
User Commands dc(1)
NAME
    dc - desk calculator
SYNOPSIS
    dc [ filename ]
DESCRIPTION
    dc is an arbitrary precision arithmetic package . Ordinarily
    it operates on decimal integers, but one may specify an
    input base, output base, and a number of fractional digits
    to be maintained. The overall structure of dc is a stacking
    (reverse Polish) calculator. If an argument is given, input
    is taken from that file until its end, then from the standard
    input.
```

这些信息来自于SunOS 5.8的联机帮助，大多数Unix系统关于dc的描述都是类似的，它说明了dc是一个计算器，它能够接收逆波兰表达式，算出表达式的值。逆波兰表达式指的是操作数在前，操作符在后，也称做后缀表达式，如要计算表达式2+3的值，它所对应的逆波兰表达式是23+，dc的输入/输出如下：

```
2
3
+
p
5
```

内部进行的操作是这样的：先将2入栈，再将3入栈，然后将栈顶的两个数出栈，计算它们的和，并将结果入栈，p是为了将栈顶元素打印出来。这样可以知道dc是一个基于栈的计算器，那么bc是什么？dc的运行条件又是如何被满足的呢？

读了bc的联机帮助就会知道，bc是dc的预处理器，它将用户输入的表达式转换成逆波兰表达式，然后通过一个称为管道(pipe)的通信程序交给dc，如图1.12所示。

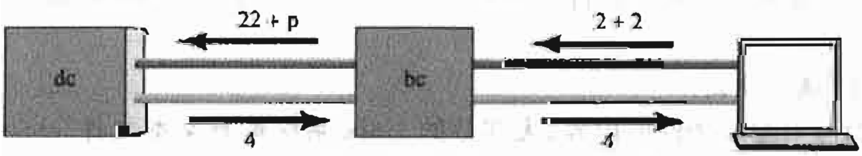


图1.12 程序交换信息

用户输入中缀表达式如“2+2”，bc将它转化为相应的后缀表达式形式，交给dc执行，dc

计算表达式的值，将结果返回给 bc，bc 再将结果以合适的形式显示在显示器上。所以对普通用户而言，bc 就是计算器。

与网络桥牌类似，计算器也是由不同的程序互相协作构成一个完整系统的，每个程序有各自的功能，互相独立、相互协作。Unix 系统编程在很多场合下，就是要解决好建立这些独立程序之间的连接和协作方式的问题。

1.5.4　从 bc/dc 到 Web

从架构的角度来看，万维网(World Wide Web)与 bc/dc 是十分类似的。通过刚才的学习可以知道 bc 负责用户界面，dc 负责后台运算，在万维网中，浏览器负责用户界面，在后面负责提供网页的是 Web 服务器，如图 1.13 所示。

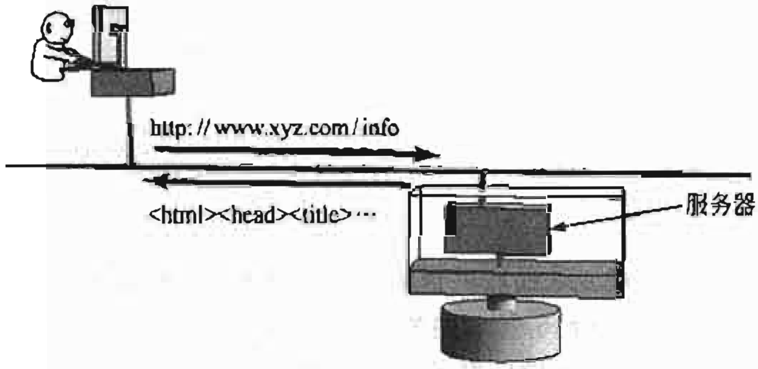


图 1.13　游览器和 Web 服务器通信

用户直接操作浏览器，从浏览器上看到网页的效果，而网页并不存放在浏览器上，而是存放在 Web 服务器上，网页由 HTML 语言写成，就像 dc 的语法一样，HTML 不是很容易理解，且不直观。用户端的工具——浏览器就像 bc 一样，从服务器上接收到信息后，会把它以容易理解的形式直观地显示给用户。

所以说万维网与 bc/dc 是类似的，而 Web 首先出现在 Unix 平台上也是一件自然而然的事情。

1.6　动 手 实 践

前面的部分通过两个问题进行学习，它们是“它能做什么?”和“它是如何实现的?”接下来应该是第三个问题了：“能不能自己编写一个?”。在本节试着自己编写一个程序来实现 more 的功能。

首先，more 能做什么?

more 可以分页显示文件的内容，大部分的 Unix 系统都有文本文件/etc/termcap，它经常被文本编辑器和游戏程序用到，用 more 来查看它的内容：

```
$ more /etc/termcap
```

more 会显示文件第一屏的内容，在屏幕的底部，more 用反白字体显示文件的百分比，这时如果按空格键，文件的下一屏内容会显示出来，如果按回车键，显示的则是下一行，如果输入"q"，结束显示，如果输入"h"，显示出来的是 more 的联机帮助。

注意，当按空格键或输入"q"后，程序会立即响应，而无需再按回车键。

more 有 3 种用法：

```
$ more filename
$ command | more
$ more < filename
```

第一种情况，more 显示文件 filename 的内容；第二种情况，more 将 command 命令的输出分页显示；第三种情况，more 从标准输入获取要分页显示的内容，而这时 more 的标准输入被重定向到文件 filename。

第二个问题，more 是如何实现的？

通过运行 more 并观察结果可以知道，more 的工作流程如下：

```
+----> show 24 lines from input
|  +--> print [more?] message
|  |      Input Enter, SPACE, or q
|  +--    if Enter, advance one line
+----     if SPACE
          if q --> exit
```

接下来要编写的程序应该像实际的 more 一样，有足够的灵活性，也就是说，如果在命令行中给出了文件名，那么就分页显示这个文件，否则的话，从标准输入得到要分页显示的内容。下面是 more 的第一个版本：

```
/* more01.c - version 0.1 of more
 * read and print 24 lines then pause for a few special commands
 */
#include <stdio.h>
#define PAGELEN 24
#define LINELEN 512
void do_more(FILE *);
int see_more();
int main( int ac, char *av[] )
{
   FILE *fp;
   if ( ac == 1 )
      do_more(stdin);
   else
      while ( --ac)
      if ( (fp = fopen( * ++av, "r" )) != NULL )
      {
```

```
            do_more( fp );
            fclose( fp );
        }
        else
            exit(1);
    return 0;
}
void do_more( FILE * fp )
/*
 * read PAGELEN lines, then call see_more() for further instructions
 */
{
    char line[LINELEN];
    int  num_of_lines = 0;
    int  see_more(), reply;
    while ( fgets( line, LINELEN, fp ) ){           /* more input */
        if ( num_of_lines == PAGELEN ) {            /* full screen? */
            reply = see_more();                     /* y: ask user */
            if ( reply == 0 )                       /* n: done */
                break;
            num_of_lines -= reply;                  /* reset count */
        }
        if ( fputs( line, stdout ) == EOF )         /* show line */
            exit(1);                                /* or die */
        num_of_lines++;                             /* count it */
    }
}
int see_more()
/*
 * print message, wait for response, return # of lines to advance
 * q means no, space means yes, CR means one line
 */
{
    int c;
    printf("\033[7m more? \033[m");                 /* reverse on a vt100 */
    while( (c=getchar()) != EOF )                   /* get response */
    {
        if ( c == 'q' )                             /* q -> N */
            return 0;
        if ( c == ' ' )                             /* ' ' => next page */
            return PAGELEN;                         /* how many to show */
        if ( c == '\n' )                            /* Enter key => 1 line */
            return 1;
    }
```

```
    return 0;
}
```

这段代码有 3 个函数，在主函数中判断应该从文件还是标准输入中获取数据，并打开相应的数据源，然后调用 do_more 函数，do_more 将数据显示在显示器上，满一屏后，调用 see_more 函数接收用户的输入，以决定下一步的动作。编译并运行上述代码：

```
$ cc more01.c - o more01
$ more01 more01.c
```

这个程序可以完成基本的功能，显示了 24 行后就停下来等待输入，而且在屏幕下方会有反白的提示 more? ，当按回车键后会显示下一行。

但是问题也是存在的，当屏幕上的文字上滚时， more? 也会随之上滚，这并不是所需要的，而且当按空格键或输入"q"后，如果不按回车键，那程序什么也不会做。

这个程序还需要改进，但是通过这个简单的例子，可以知道以下事实：Unix 编程不是很难，但也不是轻而易举的事情。

这个程序的目标很清晰，实现算法也不复杂，但是除了算法，还有其他问题需要考虑。

接下来有这样一些问题：如何使得不用回车，程序就得到输入？如何计算显示的百分比？如何去掉反白的 more? ？

先来看对数据源的处理，在 main 函数中检查命令参数的个数，如果没有参数，那就从标准输入读取数据，这样一来 more 就可以通过管道重定向来得到数据，如：

```
$ who | more
```

who 命令列出当前系统中活动的用户，管道命令"|"将 who 的输出重定向到 more 的输入，结果是每次显示 24 个用户后暂停，在有很多用户的情况下，用 more 来对 who 的输出进行分页就会很有必要。

接下来是输入重定向的问题，看以下例子：

```
$ ls /bin   more01
```

期望的结果是将/bin 目录下的文件分页，显示 24 行以后暂停。

然而实际的运行结果并不是这样的，24 行以后并没有暂停而是继续输出，问题在哪里呢？

当 more01 读入第 24 后，它打印了 more? ，然后等待用户的输入。

用户的输入是从哪里来的？在 more01 中用 getchar()，它是从标准输入读数据的，问题就在这里。刚才的命令：

```
$ ls /bin | more01
```

已经将 more01 的标准输入重定向到 ls 的标准输出，这样 more01 将从同一个数据流中读用

户的输入，这显然有问题，图 1.14 描述了这种状况。

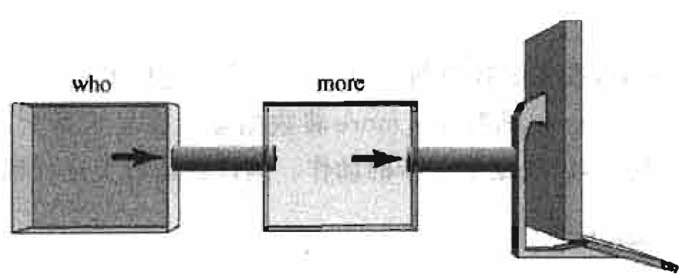


图 1.14　more 从标准输入读数据

解决这个问题的方法是，从标准输入中读入要分页的数据，直接从键盘读用户的输入，如图 1.15 所示。

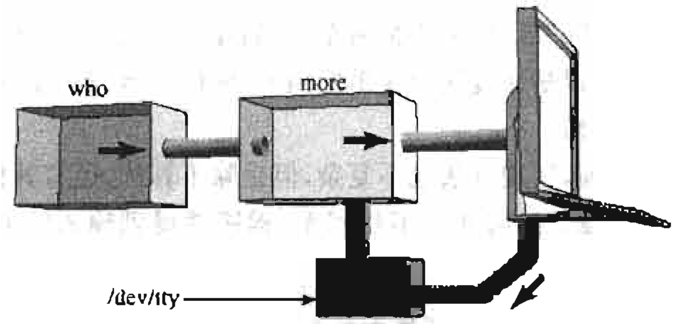


图 1.15　more 从键盘读用户的输入

图 1.15 中有一个文件/dev/tty，这是键盘和显示器的设备描述文件，向这个文件写相当于显示在用户的屏幕上，读相当于从键盘获取用户的输入。即使程序的输入/输出被“<”或“>”重定向，程序还是可以通过这个文件与终端交换数据。

从图 1.15 中可以知道，more 有两个输入，程序的标准输入是 ls 的输出，将其分页显示到屏幕上，当 more 需要用户输入时，它可以从/dev/tty 得到数据。

运用上述知识改进 more01.c，得到 more02.c：

```
/* more02.c - version 0.2 of more
 * read and print 24 lines then pause for a few special commands
 * feature of version 0.2: reads from /dev/tty for commands
 */
#include <stdio.h>
#define PAGELEN 24
#define LINELEN 512
void do_more(FILE * );
int see_more(FILE * );
int main( int ac, char * av[] )
{
    FILE * fp;
```

```
    if ( ac == 1 )
        do_more( stdin );
    else
        while ( --ac )
            if ( (fp = fopen( *++av, "r" )) != NULL )
            {
                do_more( fp );
                fclose( fp );
            }
            else
                exit(1);
    return 0;
}
void do_more( FILE *fp )
/*
 * read PAGELEN lines, then call see_more() for further instructions
 */
{
    char line[LINELEN];
    int  num_of_lines = 0;
    int  see_more(FILE *), reply;
    FILE *fp_tty;
    fp_tty = fopen( "/dev/tty", "r" );          /* NEW: cmd stream */
    if ( fp_tty == NULL )                       /* if open fails */
        exit(1);                                /* no use in running */
    while ( fgets( line, LINELEN, fp ) ){       /* more input */
        if ( num_of_lines == PAGELEN ) {        /* full screen? */
            reply = see_more(fp_tty);           /* NEW: pass FILE *  */
            if ( reply == 0 )                   /* n: done */
                break;
            num_of_lines -= reply;              /* reset count */
        }
        if ( fputs( line, stdout ) == EOF )     /* show line */
            exit(1);                            /* or die */
        num_of_lines++;                         /* count it */
    }
}
int see_more(FILE *cmd)                         /* NEW: accepts arg */
/*
 * print message, wait for response, return # of lines to advance
 * q means no, space means yes, CR means one line
 */
{
    int c;
```

```
    printf("\033[7m more? \033[m");                /* reverse on a vt100 */
    while( (c = getc(cmd)) != EOF )                /* NEW: reads from tty */
    {
        if ( c == 'q' )                            /* q -> N */
            return 0;
        if ( c == ' ' )                            /* ' ' => next page */
            return PAGELEN;                        /* how many to show */
        if ( c == '\n' )                           /* Enter key => 1 line */
            return 1;
    }
    return 0;
}
```

编译并测试新的程序：

```
$ cc -o more02 more02.c
$ ls /bin | more02
```

more02.c 可以从标准输入得到数据，也可以从键盘得到用户的输入，同时通过编写 more02.c，增加了对文件/dev/tty 的了解。

more02.c 还需要进一步完善，当用户按空格键或输入"q"后，还得按回车键，程序才会动作，而且输入的字符会显示出来。实际的 more 是不需要额外的回车的，而且输入的字符也不会回显。

3. 对输入的进一步处理

用户操作的终端有很多参数，可以调整参数使得用户输入的字符被立即送到程序，而不用等待回车，还可以使输入的字符不回显，如图 1.16 所示。

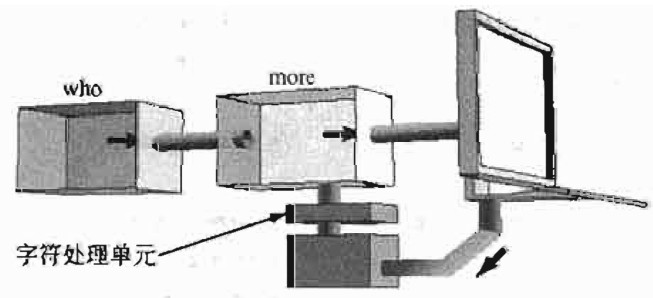


图 1.16 终端参数是可调的

图 1.16 中新加入部分是用于调整终端参数的，程序运行的时候可以动态地调整终端的参数。

要编写一个完善的 more 还有很多工作要做，以下是留给读者的问题。如何知道文件中已显示的百分比？要知道百分比就必须知道文件的大小，这些信息操作系统是提供的，需要用合适的系统调用来得到。如何反白显示文字？如何确定每一页的行数？这些都跟终端类型有关，如果将每一页定为 24 行、终端类型定为 vt100，那么程序就缺乏足够的灵活性。如

何使程序能够处理各种类型的终端？那就需要学习如何控制和调整终端参数的知识。

1.7　工作步骤与概要图

1.7.1　接下来的工作步骤

Unix系统是一个多用户系统，它允许很多用户很多程序同时工作，程序经常对文件、目录进行操作，对数据进行转换或传输。同一台机器上的不同程序之间，甚至不同机器上程序之间通过网络都可以互相通信。

这么多复杂的程序，它们的功能是什么？是如何工作的？在中间操作系统做了些什么？随着学习的深入，这些问题会越来越多地被解答。

在研究more的过程中展示了解决问题的步骤，首先分析一个实际存在的程序，弄清它的功能，分析它的实现原理，然后自己编写一个。通过对程序的不断完善来更多地了解Unix系统和它的工作原理。这也是本书采用的主要方法。

1.7.2　Unix的概要图

Unix的结构如图1.17所示。

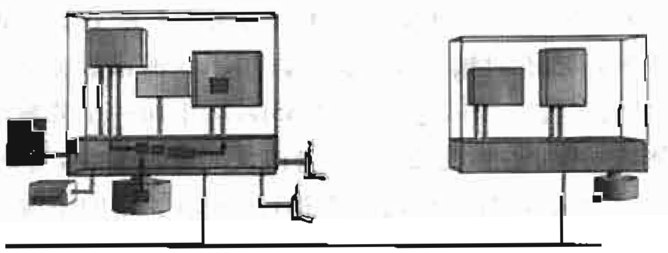


图1.17　Unix系统的主要结构

图1.17描述了Unix系统的主要结构，内存被分为系统空间和用户空间，内核和它的数据结构位于系统空间，用户程序位于用户空间，用户通过终端连接到系统，文件存放在磁盘上，各种各样的设备被内核直接管理，用户程序可以通过内核来访问设备，最后还有网络连接，用户可以通过网络接入系统。

接下来的每章都会关注系统的某一个部分，介绍相关的系统调用逻辑和数据结构。学完本书后，会对图1.17中每一个部分都有所了解，应该能够编写一般的Unix系统程序，如网络桥牌。

1.7.3　Unix的发展历程

这本书的主要内容是介绍Unix的系统结构和相关概念，以及如何编写Unix程序，你可能会问，Unix从哪里来？它是怎么发展的？在这里简要地介绍Unix的发展历程。

1969年，贝尔实验室的计算机科学家发明了Unix，最初的Unix是由内核和一些工具组成的，它并不是一个商业产品，实际上在20世纪70年代，贝尔实验室向大学和研究机构提供

Unix，包括完整的源代码，仅收取象征性的费用。贝尔实验室以及其他的计算机科学家不断地改进 Unix。到 1980 年，一些公司发布了商用的 Unix，这时的 Unix 主要分为两个系列，一个是 AT&T 公司的 System V，另一个是加利福尼亚大学伯克利分校的 BSD，大多数的 Unix 都源自上述两个 Unix 版本。几年后，伯克利停止了 BSD 的研究，System V 则被销售给了其他公司。

Unix 在商业计算和研究机构得到很大的发展，出现了不同的方向，如有些 Unix 的实时性就很出众。虽然各种 Unix 不尽相同，但 Unix 的核心架构和其主要系统调用却可以保持稳定，1980 年 AT&T 的 Unix 与 1991 年在赫尔辛基诞生的 Unix 会有很大不同，但 1980 年编写的 Unix 程序稍加修改就可以在 1991 年的 Unix 上运行。

那么什么是 Unix 呢？只要有与这些版本类似的结构和运行特性，提供应有的系统服务，那就是 Unix。由于不同机构对 Unix 的不断发展，现在有些系统看起来很像 Unix，但在代码里却看不到 System V 或 USB 的影子，如 GNU/Linux。POSIX 标准中用形式化的方法描述了 Unix 的系统接口，但是要了解 Unix，单单一个标准是不够的。

Unix 有很长的发展历史和不同的版本，所以，一本书所能涵盖的是十分有限的。本书只描述了不同的 Unix 共有的原理、技术和结构，各个不同版本的 Unix 的特点并不在本书介绍的范畴内。

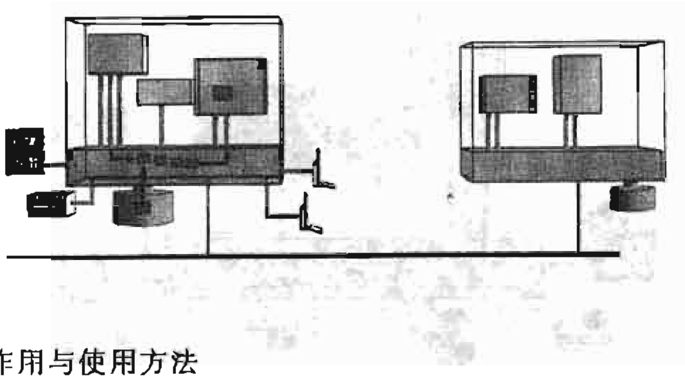
你可能工作在 SunOS 上，也可能在 AIX 或其他的 Unix 平台上，关于特定平台的知识可以参考所使用平台的联机帮助，实际上对本书的学习很大程度上要借助联机帮助。

要是注意的话就会发现，有时候一项功能可以用不同的函数来实现，这有两个原因。一是 Unix 是由不同的机构完善的，如 AT&T 和 BSD，对于同一个要解决的问题，他们采用了不同的函数名来实现。另一个原因是 Unix 自身的发展需要兼容，当有一个新的服务可以替代老的服务时，人们并不会把老的系统调用去掉，而是增加新的系统调用，这就使以前编写的程序也可以顺利地运行。在本书中也有这样的情况，可以用不同的函数来做同一件事情，作为系统程序员，这是经常的和不可避免的。

小　结

- 计算机系统中包含了很多系统资源，如硬盘、内存、外围设备、网络连接等，程序利用这些资源来对数据进行存储、转换和处理。
- 多用户系统需要一个中央管理程序，Unix 的内核就是这样的程序，它可以对程序和资源进行管理。
- 用户程序要访问设备必须经过内核。
- 一些 Unix 的系统功能是由多个程序的协作而实现的。
- 要编写系统程序，必须对系统调用和相关的数据结构有深入的理解。

第 2 章　用户、文件操作与联机帮助：编写 who 命令



概念与技巧

- 联机帮助的作用与使用方法
- Unix 的文件操作函数：open、read、write、lseek、close
- 文件的建立与读写
- 文件描述符
- 缓冲：用户级的缓冲与内核级的缓冲
- 内核模式、用户模式和系统调用的代价
- Unix 表示时间的方法与时间格式间的转换
- 借助 utmp 文件来列出已登录的用户
- 系统调用中的错误检测与处理

相关的系统调用

- open、read、write、creat、lseek、close
- perror

相关命令

- man
- who
- cp
- login

2.1 介　　绍

在使用 Unix 的时候，经常需要知道有哪些用户正在使用系统，系统是否很繁忙，某人是否正在使用系统等。为了回答这些问题，可以使用 who 命令，所有的多用户系统都会有这个命令。这个命令会显示系统中活动用户的情况。接下来的问题是，who 命令是如何工作的呢？

本章分析 Unix 中的 who 命令，通过分析来学习 Unix 的文件操作。除此之外，还将学习如何从 Unix 的联机帮助中得到有用的信息。

2.2 关于命令 who

下面给出了 Unix 系统的概览图，如图 2.1 所示。

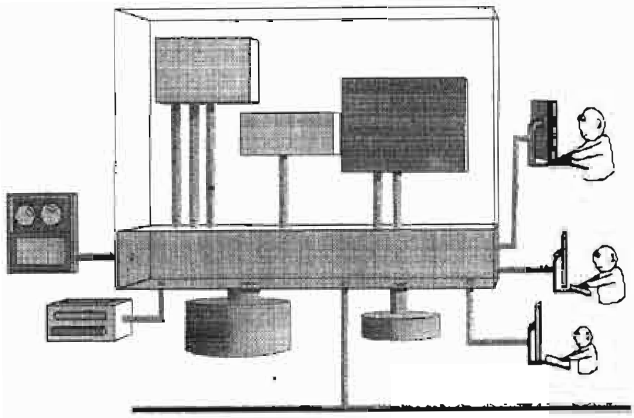


图 2.1 用户、文件、进程、设备和内核

图 2.1 中最大的长方体代表计算机内存，它被分为用户空间和系统空间，用户通过终端连接到系统，一大一小两个柱状体代表两个硬盘，系统中还有一个打印机。靠上方的 3 个较小的长方体代表 3 个应用程序，它们运行在用户空间，通过内核与外界进行通信，应用程序和内核之间的连线代表通信管道。

本章的第一个程序通过对以下 3 个问题的解答来学习 who 命令：

1. who 命令能做些什么?
2. who 命令是如何工作的?
3. 如何编写 who?

命令也是程序

在开始之前，需要声明的是，在 Unix 系统中，几乎所有的命令都是人为编写的程序，如 who 和 ls，而且它们中的大多数都是用 C 语言写的。当在命令行中输入 ls，Shell(命令解释器)就知道你想运行名字为 ls 的程序。如果对 ls 所提供的功能还不满意，完全可以编写和使用自己的 ls 命令。

在 Unix 系统中增加新的命令是一件很容易的事。把程序的可执行文件放到以下任意一个目录就可以了：/bin、/usr/bin、/usr/local/bin，这些目录里面存放着很多系统命令。Unix 系统中一开始并没有这么多的命令，一些人编写程序用来解决某个特定的问题，而其他人也觉得这个程序很有用，随着越来越多的使用，这个程序就逐渐成了 Unix 的标准命令。说不定哪一天，你编写的程序也会成为标准命令。

2.3　问题 1：who 命令能做些什么

如果想要知道都有谁正在使用使用系统，只有输入 who 命令，输出如下：

```
$ who
heckerl   ttyp1 Jul 21 19:51 (tide75.surfcity.com)
nlopez    ttyp2 Jul 21 18:11 (roam163-141.student.ivy.edu)
dgsulliv  ttyp3 Jul 21 14:18 (h004005a8bd64.ne.mediaone.net)
ackerman  ttyp4 Jul 15 22:40 (asdl-254.fas.state.edu)
wwchen    ttyp5 Jul 21 19:57 (circle.square.edu)
barbier   ttyp6 Jul  8 13:08 (labpc18.elsie.special.edu)
ramakris  ttyp7 Jul 13 08:51 (roam157-97.student.ivy.edu)
czhu      ttyp8 Jul 21 12:47 (spa.sailboat.edu)
bpsteven  ttyp9 Jul 21 18:26 (207.178.203.99)
molay     ttypa Jul 21 20:00 (xyz73-200.harvard.edu)
$
```

每一行代表一个已经登录的用户，第 1 列是用户名，第 2 列是终端名，第 3 列是登录时间，第 4 列是用户的登录地址，某些版本的 who 命令在默认状态下不给出第 4 列的内容。

阅读手册

通过直接运行命令，可以了解 who 的大致功能，要进一步了解 who 的用法，需要借助联机帮助。

每一个 Unix 在发售的时候都会带上大量的有关各个命令使用方法的文档。以前，这些文档是打印出来的，现在有电子版的可供使用。查看联机帮助的命令是 man，如要查看 who 的帮助，可输入：

```
$ man who
who(1)
NAME
  who - Identifies users currently logged in
SYNOPSIS
  who [-a] |[-AbdhHlmMpqrstTu] [file]
  who am i
  who am I
  whoami
  The who command displays information about users and processes on the local system.

STANDARDS

  Interfaces documented on this reference page conform to industry standards as follows:
  who: XPG4, XPG4-UNIX
```

```
    Refer to the standards(5) reference page for more information about industry standards and
    associated tags.

OPTIONS

    -a  Specifies all options; processes /var/adm/utmp or the named file with all options on.
        Equivalent to using the -b, -d, -l, -p, -r, -t, -T, and -u options.
more(10%)
```

所有命令的联机帮助都有相同的基本格式，从第1行可以知道这是关于哪个命令的帮助，还可以知道这个帮助是位于哪一节的。在这个例子中，从第1行的内容 who(1)，可以知道这是 who 命令的帮助，它的小节编号是1。Unix 的联机帮助分为很多节，如第1小节中是关于用户命令的帮助，第2小节中是关于系统调用的帮助，第5小节中是关于配置文件的帮助。你可查看一下 Unix 系统的联机帮助，了解其他节讲述的内容。

名字(NAME)部分包含命令的名字以及对这个命令的简短说明。

概要(SYNOPSYS)部分给出了命令的用法说明，包括命令格式、参数和选项列表。选项指的是一个短线后面紧跟着一个或多个英文字母，如 -a、-Bc，命令的选项影响该命令所进行的操作。

在联机帮助中，方括号([-a])表示该选项不是一个必须的部分。帮助中指出 who 的写法可以是 who，或者 who -a，或者 who - 加上 AbdhHlmMpqrstTu 这些字母的任意组合，在命令的末尾还可以有一个文件参数。

从帮助中可以知道 who 命令还有其他3种形式：

```
who am i
who am I
whoami
```

从联机帮助中还可以获得上述形式的进一步帮助。

描述(DESCRIPTION)部分是关于命令功能的详细阐述，根据命令和平台的不同，描述的内容也不同，有的简洁、精确，有的包含了大量的例子。不管怎么样，它描述了命令的所有功能，而且是这个命令的权威性解释。

选项(OPTIONS)部分给出了命令行中每一个选项的说明。早期的 Unix 命令的功能都很简单，每个命令只有一两个选项，但随着时间的推移，命令的功能越来越多，基本上每个选项用来实现一个功能，所以选项也越来越多，像 who 命令就有很多选项。

参阅(SEE ALSO)部分包含与这个命令相关的其他主题。有些帮助还有 BUG 部分。

2.4 问题2：who 命令是如何工作的

前面看到 who 命令可以显示出当前系统中已经登录的用户信息，联机帮助中描述了 who 的功能和用法，现在的问题是：who 是如何来实现这些功能的？

你可能会认为，像 who 这样的系统程序一定会用到一些特殊的系统调用，需要高级管理员的权限，要编写这样的程序得要花很多钱来购买系统开发工具，包括光盘、参考书等。

实际上，所需的资料都在系统中，你要知道的仅仅是如何找到这些资料。

1. 从 Unix 中学习 Unix

以下 4 项技巧会有助于你的学习：

- 阅读联机帮助
- 搜索联机帮助
- 阅读.h 文件
- 从参阅部分(SEE ALSO)得到启示

2. 阅读联机帮助

以 who 为例，在命令行输入如下命令：

```
$ man who
```

翻到描述部分，如果是在 SunOS 平台上，可以看到如下内容：

```
DESCRIPTION
    The who utility can list the user's name, terminal line, login time, elapsed time since
    activity occurred on the line, and the process - ID of the command interpreter (shell) for
    each current UNIX system user. It examines the /var/adm/utmp file to obtain its information.
    If file is given, that file (which must be in utmp(4) format) is examined. Usually, file will
    be /var/adm/wtmp, which contains a history of all the logins since the file was last created.
```

上述描述说明，已登录用户的信息是放在文件/var/adm/utmp 中的，who 通过读该文件获得信息。可以通过搜索联机帮助来了解这个文件的结构信息。

3. 搜索联机帮助

使用带有选项－k 的 man 命令可以根据关键字搜索联机帮助。如果要查找"utmp"的信息，在命令行输入如下命令：

```
$ man - k utmp
endutent      getutent (3c)    - access utmp file entry
endutxent     getutxent (3c)   - access utmpx file entry
getutent      getutent (3c)    - access utmp file entry
getutid       getutent (3c)    - access utmp file entry
getutline     getutent (3c)    - access utmp file entry
getutmp       getutxent (3c)   - access utmpx file entry
getutmpx      getutxent (3c)   - access utmpx file entry
getutxent     getutxent (3c)   - access utmpx file entry
getutxid      getutxent (3c)   - access utmpx file entry
getutxline    getutxent (3c)   - access utmpx file entry
pututline     getutent (3c)    - access utmp file entry
pututxline    getutxent (3c)   - access utmpx file entry
setutent      getutent (3c)    - access utmp file entry
```

```
setutxent    getutxent (3c)  - access utmpx file entry
ttyslot      ttyslot (3c)    - find the slot in the utmp file of the current user
updwtmp      getutxent (3c)  - access utmpx file entry
updwtmpx     getutxent (3c)  - access utmpx file entry
utmp         utmp (4)        - utmp and wtmp entry formats
utmp2wtmp    acct (1m)       - overview of accounting and miscellaneous accounting
                               commands
utmpd        utmpd (1m)      - utmp and utmpx monitoring daemon
utmpname     getutent (3c)   - access utmp file entry
utmpx        utmpx (4)       - utmpx and wtmpx entry formats
utmpxname    getutxent (3c)  - access utmpx file entry
wtmp         utmp (4)        - utmp and wtmp entry formats
wtmpx        utmpx (4)       - utmpx and wtmpx entry formats
$
```

以上的输出是在 SunOS 平台上的输出，不同版本的 Unix 输出可能会有所不同，其中每一行都包含帮助的主题、标题和一段简短的描述。注意这一行：

```
utmp    utmp (4)       - utmp and wtmp entry formats
```

看起来好像就是所需要的。这一行里的"4"是小节编号，说明该帮助是位于第 4 节，在查看该帮助内容的时候，注意别把小节编号漏了：

```
$ man 4 utmp
utmp(4)
NAME
  utmp, wtmp - Login records
SYNOPSIS
  #include <utmp.h>
DESCRIPTION
  The utmp file records information about who is currently using the system.
  The file is a sequence of utmp entries, as defined in struct utmp in the utmp.h file.

  The utmp structure gives the name of the special file associated with the user's terminal, the
  user's login name, and the time of the login in the form of time(3), The ut_type field is the type
  of entry, which can specify several symbolic constant values. The symbolic constants are
  defined in the utmp.h file.

  The wtmp file records all logins and logouts. A null user name indicated a logout on the
  associated terminal. A terminal referenced with a tilde (~) indicates that the system was
  rebooted at the indicated time. The adjacent pair of entries with terminal names referenced by a
  vertical bar (|) or a right brace (}) indicate the system - maintained time just before and just
  after a date command has changed the system's time frame.
  The wtmp file is maintained by login(1) and init(8). Neither of these programs creates the
  file, so, if it is removed, record keeping is turned off, See ac(8) for information on the file.
```

```
FILES
  /usr/include/utmp.h
  /var/adm/utmp
more(88 % )
```

到此已经离目标不远了，who 的联机帮助说明 who 要读 utmp 这个文件，进一步，从以上的说明可以知道 utmp 这个文件里面保存的是结构数组，数组元素是 utmp 类型的结构，可以在 utmp. h 中找到 utmp 类型的定义，接下来的问题是：utmp. h 这个文件在哪里？

阅读以上帮助，可以在 FILES 部分找到 utmp. h 这个文件的位置，在/usr/include 目录里。

在进行下一步(分析. h 文件)之前，还有些东西要引起注意，上述帮助提到 wtmp 这个文件记录了关于登录和注销的信息，它涉及到以下几个命令：login(1)、init(8)和ac(8)，这些命令将在以后的章节讲到。

通过联机帮助来学习 Unix 就像在网络上寻找信息一样，经常能从某一个帮助主题中找到相关信息，链接到其他有用或有趣的主题。言归正传，接下来分析<utmp. h>这个文件。

4. 阅读. h 文件

从 utmp 的联机帮助中可以知道，utmp 中的数据结构定义在/usr/include/utmp. h 中。在 Unix 系统中，大多数的头文件都存放在/usr/include 这个目录里，当 C 语言编译器在源程序中发现如下的定义：

```
#include <stdio.h>
```

它会到/usr/include 中寻找相应的头文件。接下来用 more 命令来查看这个文件的内容：

```
$ more /usr/include/utmp.h
  ...
#define UTMP_FILE      "/var/adm/utmp"
#define WTMP_FILE      "/var/adm/wtmp"
#include <sys/types.h>             /* for pid_t, time_t */
/*
 * Structure of utmp and wtmp files.
 *
 * Assuming these numbers is unwise.
 */
#define ut_name ut_user            /* compatibility */
struct utmp {
        char ut_user[32];          /* User login name */
        char ut_id[14];            /* /etc/inittab id - IDENT_LEN in init */
        char ut_line[32];          /* device name(console, lnxx) */
        short ut_type;             /* type of entry */
        pid_t ut_pid;              /* process id */
        struct exit_status {
           short e_termination;    /* Process termination status */
```

```
            short e_exit;           /* Process exit status */
        } ut_exit;                  /* The exit status of a process marked as DEAD_PROCESS */
        time_t ut_time;             /* Time entry was made */
        char ut_host[64];           /* Host name same as MAXHOSTNAMELEN */
};
/* Definitions for ut_type */
utmp.h(60%)
```

略过所有介绍性的内容，直接来看 utmp 结构所保存的登录记录。它包含 8 个成员变量，ut_user 数组保存登录名，ut_line 数组保存设备名，也就是用户的终端类型，ut_time 保存登录时间，ut_host 保存用户用于登录的远程计算机的名字。

utmp 这个结构所包含的其他成员没有被 who 命令所用到。

各个平台上的 utmp 结构可能不会完全相同，具体的内容由 utmp.h 来决定。从成员变量来看，其中的绝大部分字段在大多数的 Unix 平台上是相同的，被标记为兼容(compatibility)的行更可能出现不同。通常头文件中的注释提供了一些有用的信息。

who 的工作原理

通过阅读 who 和 utmp 的联机帮助，以及头文件/usr/include/utmp.h，可以知道 who 的工作原理，who 通过读文件来获得需要的信息，而每个登录的用户在文件中都有对应的记录。who 的工作流程可以用图 2.2 来表示。

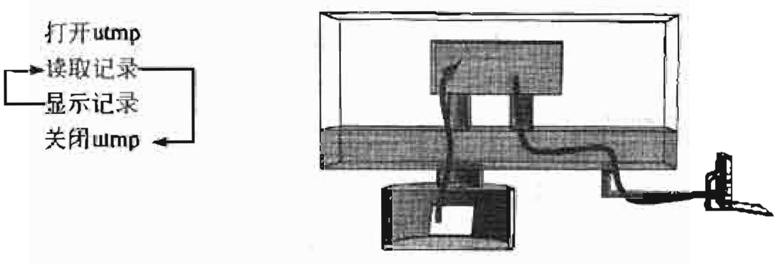


图 2.2　who 命令的数据流

文件中的结构数组存放登录用户的信息，所以直接的想法就是把记录一个一个地读出并显示出来，是不是就这么简单呢？

虽然没有看过 who 的源代码，但从联机帮助中可以了解 who 要完成的功能及实现原理，所涉及的数据结构的信息也可以从头文件中获取。接下来是实践的时候了。

2.5　问题 3：如何编写 who

接下来要编写自己的 who 程序，为了能够顺利完成，需要经常从联机帮助中获取信息。测试程序时，要把程序的输出与系统 who 命令的输出做比较。通过分析可以确认，在编写 who 程序时只有两件事情是要做的：

- 从文件中读取数据结构
- 将结构中的信息以合适的形式显示出来

2.5.1　问题：如何从文件中读取数据结构

可以调用 getc 和 fgets 函数从文件中读字符或字符串，但是如何读出数据结构中的信息呢？当然可以用 getc 逐个字节地读取，但这样太繁琐，而且效率很低。要找一种可以一次读出整个数据结构的方法。

还是到联机帮助中寻找答案，可以找那些与 file 和 read 都有关的帮助，但是 man 命令的选项 -k（根据关键字查找）只支持一个关键字的查找。在我的系统中，与 file 相关的主题有 537 个，如果一个一个地来看，这就太慢了，可以借助 Unix 命令 grep 来从这 537 个主题中查找与 read 相关的主题：

```
$ man -k file | grep read
_llseek (2)         - reposition read/write file offset
fileevent (n)       - Execute a script when a channel becomes readable or writable
gftype (1)          - translate a generic font file for humans to read
lseek (2)           - reposition read/write file offset
macsave (1)           Save Mac files read from standard input
read (2)            - read from a file descriptor
readprofile (1)     - a tool to read kernel profiling information
                    scr_dump, scr_restore, scr_init, scr_set (3) - read (write) a curses screen
                    from (to) a file
tee (1)             - read from standard input and write to standard output and files
$
```

其中最有可能的是 read(2)，其他的看起来都不像，所以进一步地看 read(2)的帮助：

```
$ man 2 read
READ(2)                    System calls                    READ(2)
NAME
  Read - read from a file descriptor
SYNOPSIS
   #include <unistd.h>
  ssize_t read(int fd, void * buf, size_t count);
DESCRIPTION
  read() attempts to read up to count bytes from file descriptor fd into the buffer starting
  at buf.
  If count is zero, read() returns zero and has no other results. If count is greater than SSIZE_
  MAX, the result is unspecified.
RETURN VALUE
  On success, the number of bytes read is returned (zero indicates end of file), and the file
  position is advanced by this number. It is not an error if this number is smaller than the number
  of bytes requested; this may happen for example because fewer bytes are actually available
```

```
right now(maybe because we were close to end - of - file, or because we are reading from a pope,
or from a terminal), or because read() was interrupted by a signal. On error, - 1 is returned,
and errno is set appropriately. In this case it is left unspecified whether the file position(if
any) changes.
```

这个系统调用可以将文件中一定数目的字节读入一个缓冲区，因为每次都要读入一个数据结构，所以要用 sizeof(struct utmp)来指定每次读入的字节数。read 函数需要一个文件描述符作为输入参数，如何得到文件描述符呢？在 read 的联机帮助中的最后部分有以下描述：

```
RELATED INFORMATION(called SEE ALSO in some versions)
  Functions: fcntl(2), creat(2), dup(2), ioctl(2), getmsg(2), lockf(3), lseek(2), mtio(7),
  open(2), pipe(2), poll(2), socket(2), socketpair(2), termios(4),streamio(7),opendir(3),
  lockf(3)
  Standards: standards(5)
```

其中包含对 open(2)的引用，在命令行输入：

```
$ man 2 open
```

查看 open 的联机帮助，从 open 中又可以找到对 close 的引用，通过阅读联机帮助，可以知道以上 3 个系统调用都是进行文件操作所必需的。

2.5.2　答案：使用 open、read 和 close

使用上述 3 个系统调用可以从 utmp 文件中取得用户登录信息，这 3 个系统调用的联机帮助会包含很多内容，尤其是应用于管道、设备和其他数据源时，会涉及很多不常用的选项以及复杂的用法，但在这里，只需要关心它们最基本的用法。

1. 打开一个文件：open

这个系统调用在进程和文件之间建立一条连接，这个连接被称为文件描述符，它就像一条由进程通向内核的管道，如图 2.3 所示。

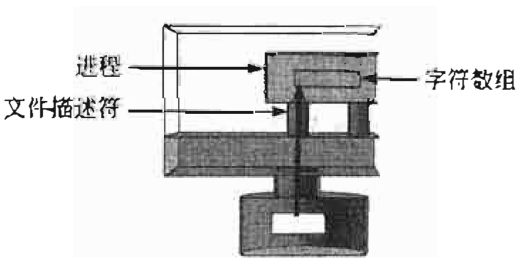


图 2.3　文件描述符是对文件的连接

open 的基本用法如下。

open		
目标	打开一个文件	
头文件	#include <fcntl.h>	
函数原型	int fd = open(char * name, int how)	
参数	name	文件名
	how	O_RDONLY, O_WRONLY, or O_RDWR
返回值	−1	遇到错误
	int	成功返回

要打开一个文件，必须指定文件名和打开模式，有 3 种打开模式：只读、只写、可读可写，分别对应于 O_RDONLY、O_WRONLY、O_RDWR，这在头文件/usr/include/fcntl.h 中有定义。

打开文件是内核提供的服务，如果在打开过程中内核检测到任何错误，这个系统调用就会返回−1。

错误类型是各种各样的，如：要打开的文件不存在。即使文件存在，也可能因为权限不够而无法打开，或者是无权访问文件所在的目录。在 open 的联机帮助中列出了各种可能的错误，本章结尾还会进一步讨论出错处理的问题。

当一个文件已经被打开，是否允许再次打开呢？这种情况发生在有多个进程要同时访问一个文件的时候。Unix 并不禁止一个文件同时被多个进程访问，如果禁止的话，那两个用户就无法同时使用 who 命令了。

如果文件被顺利打开，内核会返回一个正整数的值，这个数值就叫做文件描述符。刚才讲过，打开文件会建立进程和文件之间的连接，文件描述符就是用来惟一标识这个连接的，如果同时打开好几个文件，它们所对应的文件描述符是不同的，如果将一个文件打开多次，对应的文件描述符也不相同。

必须通过文件描述符对文件进行操作。

2. 从文件读取数据：read

通过 read 函数来读取数据，read 的用法如下：

read		
目标	把数据读取到缓冲区	
头文件	#include <unistd.h>	
函数原型	ssize_t numread = read(int fd, void * buf, size_t qty)	
参数	fd	文件描述符
	buf	用来存放数据的目的缓冲区
	qty	要读取的字节数
返回值	−1	遇到错误
	numread	成功读取

read 这个系统调用请求内核从 fd 所指定的文件中读取 qty 字节的数据，存放到 buf 所指定的内存空间中，内核如果成功地读取了数据，就返回所读取的字节数目，否则返

回－1。

这里有个问题，就是最终读到的数据可能没有你所要求的那么多，为什么呢？可能是因为文件中剩余的数据没有要求的那么多。例如：程序要求读 1000 字节的数据，而文件的长度才 500 个字节，那么程序就只能读到 500 字节。当读到文件末尾时再要读的话，numread 会是 0，因为已经没有数据可读了。

调用 read 函数时可能会遇到的错误在联机帮助中有详细的描述。

3. 关闭文件：close

当不需要再对文件进行读写操作时，就要把文件关闭。close 的用法如下。

close	
目标	关闭一个文件
头文件	#include <unistd.h>
函数原型	int result = close(int fd)
参数	fd　　文件描述符
返回值	−1　　遇到错误 0　　成功关闭

close 这个系统调用会关闭进程和文件 fd 之间的连接，如果关闭的过程中出现错误，close 返回－1，例如：fd 所指的文件并不存在。其他可能遇到的错误在联机帮助中有详细的描述。

2.5.3 编写 who1.c

到目前为止，已经离目标很近了，明白了 who 的工作原理，知道了要先打开文件，然后读数据，最后关闭文件，以及上述操作所需要的系统调用，这样可以编写如下代码：

```
/* who1.c - a first version of the who program
 *          open, read UTMP file, and show results
 */
#include <stdio.h>
#include <utmp.h>
#include <fcntl.h>
#include <unistd.h>

#define SHOWHOST                    /* include remote machine on output */

int main()
{
    struct utmp   current_record;   /* read info into here */
    int           utmpfd;           /* read from this descriptor */
    int           reclen = sizeof(current_record);

    if ( (utmpfd = open(UTMP_FILE, O_RDONLY)) == -1 ){
        perror( UTMP_FILE );        /* UTMP_FILE is in utmp.h */
```

```
        exit(1);
    }

    while ( read(utmpfd, &current_record, reclen) == reclen )
        show_info(&current_record);
    close(utmpfd);
    return 0;                       /* went ok */
}
```

这段代码应用了前面学到的内容，在 while 循环内从文件中逐条地把数据读取出来，存放在记录 current_record 中，然后调用函数 show_info 把登录信息显示出来，当文件中已经没有数据时，循环结束，最后关闭文件返回。

这里调用了函数 perror，这是一个系统函数，使用这个函数来处理系统报错，关于错误处理在本章结尾还会讨论。

2.5.4 显示登录信息

下面是 show_info 的第一个版本，它的功能是显示 utmp 记录的内容：

```
/*
 * show info()
 * displays contents of the utmp struct in human readable form
 * * note * these sizes should not be hardwired
 */
show_info( struct utmp * utbufp )
{
    printf("% -8.8s", utbufp->ut_name);     /* the logname  */
    printf(" ");                            /* a space */
    printf("% -8.8s", utbufp->ut_line);     /* the tty */
    printf(" ");                            /* a space */
    printf("% 10ld", utbufp->ut_time);      /* login time */
    printf(" ");                            /* a space */
#ifdef SHOWHOST
    printf("(% s)", utbufp->ut_host);       /* the host */
#endif
    printf("\n");                           /* newline */
}
```

在使用 printf 函数输出时，使用了定宽度的格式，这样做是为了与系统的 who 命令的输出一致，ut_time 字段是以 long int 的格式输出的，在头文件中这个字段被定义为 time_t 类型，但到目前还不知道 time_t 类型的数据应如何处理。

将上述代码编译、运行：

```
$ cc who1.c -o who1
$ who1
```

```
          system b  952601411 ()
          run - leve 952601411 ()
                    952601416 ()
                    952601416 ()
                    952601417 ()
                    952601417 ()
                    952601419 ()
                    952601419 ()
                    952601423 ()
                    952601566 ()
LOGIN     console   952601566 ()
          ttyp1     958240622 ()
shpyrko   ttyp2     964318862 (nas1 - 093.gas.swamp.org)
acotton   ttyp3     964319088 (math - guest04.williams.edu)
          ttyp4     964320298 ()
spradlin  ttyp5     963881486 (h002078c6adfb.ne.rusty.net)
dkoh      ttyp6     964314388 (128.103.223.110)
spradlin  ttyp7     964058662 (h002078c6adfb.ne.rusty.net)
king      ttyp8     964279969 (blade - runner.mit.edu)
berschba  ttyp9     964188340 (dudley.learned.edu)
rserved   ttypa     963538145 (gigue.eas.ivy.edu)
dabel     ttypb     964319455 (roam193 - 27.student.state.edu)
          ttypc     964319645 ()
rserved   ttypd     963538287 (gigue.eas.ivy.edu)
dkoh      ttype     964298769 (128.103.223.110)
          ttypf     964314510 ()
molay     ttyq0     964310621 (xyz73 - 200.harvard.edu)
          ttyq1     964311665 ()
          ttyq2     964310757 ()
          ttyq3     964304284 ()
          ttyq4     964305014 ()
          ttyq5     964299803 ()
          ttyq6     964219533 ()
          ttyq7     964215661 ()
cweiner   ttyq8     964212019 (roam175 - 157.student.stats.edu)
          ttyqa     964277078 ()
          ttyq9     964231347 ()
$
```

将上述输出与系统 who 命令的输出做对比：

```
$ who
shpyrko   ttyp2 Jul 22 22:21 (nas1 - 093.gas.swamp.edu)
acotton   ttyp3 Jul 22 22:24 (math - guest04.Williams.edu)
spradlin  ttyp5 Jul 17 20:51 (h002078c6adfb.ne.rusty.net)
```

```
dkoh      ttyp6 Jul 22 21:06 (128.103.223.110)
spradlin  ttyp7 Jul 19 22:04 (h002078c6adfb.ne.rusty.net)
king      ttyp8 Jul 22 11:32 (blade-runner.mit.edu)
berschba  ttyp9 Jul 21 10:05 (dudley.learned.edu)
rserved   ttypa Jul 13 21:29 (gigue.eas.ivy.edu)
dabel     ttypb Jul 22 22:30 (roam193-27.student.state.edu)
rserved   ttypd Jul 13 21:31 (gigue.eas.harvard.edu)
dkoh      ttype Jul 22 16:46 (128.103.223.110)
molay     ttyq0 Jul 22 20:03 (xyz73-200.harvard.edu)
cweiner   ttyq8 Jul 21 16:40 (roam175-157.student.stats.edu)
$
```

自己编写的 who 已经可以工作了，它能正确显示出用户名、终端名、远程主机名，但跟系统的 who 比起来还不完善，至少在两处有问题。

需要改进的：

- 消除空白记录
- 正确显示登录时间

2.5.5　编写 who2.c

针对版本 1 的两个问题，继续编写 who 的第 2 个版本，解决问题的方法还是通过阅读联机帮助和头文件。

1. 消除空白记录

系统所带的 who 只列出已登录用户的信息，而刚才编写的 who 1 除了会列出已登录的用户，还会显示其他的信息，而这些都来自于 utmp 文件。实际上 utmp 包含所有终端的信息，甚至那些尚未被用到的终端的信息也会存放在 utmp 中，所以要修改刚才的程序，做到能够区分出哪些终端对应活动的用户。如何区分呢？

一种简单的思路是过滤掉那些用户名为空的记录，但这样做是有问题的，如刚才的输出中，用户名为 LOGIN 的那一行对应的是控制台，而不是一个真实的用户。最好有一种方法能够指出某一条记录确实对应着已登录的用户。

在/usr/include/utmp.h 中，有以下内容：

```
/* Definitions for ut_type */

#define EMPTY           0
#define RUN_LVL         1
#define BOOT_TIME       2
#define OLD_TIME        3
#define NEW_TIME        4
#define INIT_PROCESS    5    /* Process spawned by "init" */
#define LOGIN_PROCESS   6    /* A "getty" process waiting for login */
#define USER_PROCESS    7    /* A user process */
#define DEAD_PROCESS    8
```

utmp 结构中有一个成员 ut_type，当它的值为 7(USER_PROCESS)时，表示这是一个已经登录的用户。根据这一点，对原来的程序做以下修改，就可以消除空白行：

```
show_info( struct utmp * utbufp )
{
   if ( utbufp->ut_type != USER_PROCESS )         /* users only */
         return;
   printf("% - 8.8s", utbufp->ut_name);            /* the username */
```

2. 以可读的方式显示登录时间

接下来要处理的是时间显示的问题，要把时间以易于理解的形式显示。还是需要借助联机帮助和头文件。在联机帮助中关于时间的主题很多，在命令行输入：

```
$ man - k time
```

会返回很多条记录，我曾在某个 Unix 系统上得到 73 条记录，而在另一个 Unix 系统上得到了 97 条。当然可以瞪着眼珠一条一条地看，不过用 Unix 自带的工具来过滤出有用的东西是一个更好的方法。以下是两种过滤方法：

```
$ man - k time | grep transform
$ man - k time | grep - i convert
```

很多记录都涉及到/usr/include/time.h 这个头文件，这里面有很多有用的信息。

(1) Unix 存储时间的方式：time_t 数据类型

Unix 中时间是用一个整数来表示的，它的数值是从 1970 年 1 月 1 日 0 时开始所经过的秒数，在头文件 time.h 中有以下内容：

```
typedef long int time_t;
```

存储时间的结构 time_t 实际上就是 long int。

(2) 将 time_t 显示出来：ctime

ctime 将表示时间的整数值转换成人们日常所使用的时间形式。在联机帮助的第 3 节有 ctime 的详细说明：

```
$ man 3 ctime
CTIME(3)              Linux Programmers Manual                     CTIME(3)

NAME
  asctime, ctime, gmtime, localtime, mktime - transform binary date and time to ASCII

SYNOPSIS
  #include <time.h>
  char * asctime(const struct tm * timeptr);
  char * ctime(const time_t * timep);
  struct tm * gmtime(const time_t * timep);
  struct tm * localtime(const time_t * timep);
```

```
  time_t mktime(struct tm * timeptr);

  extern char * tzname[2];
  long int timezone;
  extern int daylight;

DESCRIPTION
  The ctime(), gmtime() and localtime() functions all take an argument of data type time_t which
  represents calendar time. When interpreted as an absolute time value, it represents the number
  of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC).
...
  The ctime() function converts the calendar time timep into a string of the form
                  "Wed Jun 30 21:49:08 1993\n"

  The abbreviations for the days of the week are Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The
  abbreviations for the months are Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec.
  The return value points to a statically allocated string which might be overwritten by
  subsequent calls to any of the date and time functions. The function also
```

这正是所需要的，在 utmp 结构中有一个 time_t 类型的数据，而最终需要的是类似于以下的输出：

```
Jun 30 21:49
```

ctime(3)函数要输入一个指向 time_t 的指针，返回的时间字符串类似于以下格式：

```
Wed Jun 30 21:49:08 1993\n
    ^^^^^^^^^^^^
```

注意：并不是所有的字符串内容都需要，需要的是其中用"^"标识出来的部分，接下来就很容易处理了，将 ctime 返回的字符串从第 4 个字符开始，输出 12 个字符：

```
printf("%12.12s",ctime(&t) + 4)
```

3. 把刚才学的两点综合起来

现在明白了如何消除空白记录和如何正确地显示 ut_time 中的时间值，可以着手重新编写 who2.c 如下：

```
/* who2.c  - read /etc/utmp and list info therein
 *         - suppresses empty records
 *         - formats time nicely
 */
#include    <stdio.h>
#include    <unistd.h>
#include    <utmp.h>
#include    <fcntl.h>
#include    <time.h>
```

```
/* #define SHOWHOST */

void showtime(long);
void show_info(struct utmp * );

int main()
{
    struct utmp  utbuf;     /* read info into here */
    int          utmpfd;    /* read from this descriptor */

    if ( (utmpfd = open(UTMP_FILE, O_RDONLY)) == -1 ){
        perror(UTMP_FILE);
        exit(1);
    }

    while( read(utmpfd, &utbuf, sizeof(utbuf)) == sizeof(utbuf) )
        show_info( &utbuf );
    close(utmpfd);
    return 0;
}
/*
 * show info()
 *          displays the contents of the utmp struct
 *          in human readable form
 *          * displays nothing if record has no user name
 */
void show_info( struct utmp * utbufp )
{
    if ( utbufp->ut_type != USER_PROCESS )
        return;

    printf("% -8.8s", utbufp->ut_name);          /* the logname */
    printf(" ");                                 /* a space */
    printf("% -8.8s", utbufp->ut_line);          /* the tty */
    printf(" ");                                 /* a space */
    showtime( utbufp->ut_time );                 /* display time */
#ifdef SHOWHOST
    if ( utbufp->ut_host[0] != '\0' )
        printf(" (%s)", utbufp->ut_host);        /* the host */
#endif
        printf("\n");                            /* newline */
}

void showtime( long timeval )
/*
 * displays time in a format fit for human consumption
```

```
 * uses ctime to build a string then picks parts out of it
 * Note: %12.12s prints a string 12 chars wide and LIMITS
 * it to 12chars.
 */
{
    char *cp                        /* to hold address of time */
    cp = ctime(&timeval);           /* convert time to string */
                                    /* string looks like */
                                    /* Mon Feb 4 00:46:40 EST 1991 */
                                    /* 0123456789012345. */
    printf("%12.12s", cp + 4 );     /* pick 12 chars from pos 4 */
}
```

4. 测试 who2.c

为了便于对比，先关掉 SHOWHOST 这个选项，编译 who2.c 将其结果与系统所带的 who 进行比较：

```
$ cc who2.c -o who2
$ who2
rlscott    ttyp2 Jul 23 01:07
acotton    ttyp3 Jul 22 22:24
spradlin   ttyp5 Jul 17 20:51
spradlin   ttyp7 Jul 19 22:04
king       ttyp8 Jul 22 11:32
berschba   ttyp9 Jul 21 10:05
rserved    ttypa Jul 13 21:29
rserved    ttypd Jul 13 21:31
molay      ttyq0 Jul 22 20:03
cweiner    ttyq8 Jul 21 16:40
mnabavi    ttyx2 Apr 10 23:11

$ who
rlscott    ttyp2 Jul 23 01:07
acotton    ttyp3 Jul 22 22:24
spradlin   ttyp5 Jul 17 20:51
spradlin   ttyp7 Jul 19 22:04
king       ttyp8 Jul 22 11:32
berschba   ttyp9 Jul 21 10:05
rserved    ttypa Jul 13 21:29
rserved    ttypd Jul 13 21:31
molay      ttyq0 Jul 22 20:03
cweiner    ttyq8 Jul 21 16:40
mnabavi    ttyx2 Apr 10 23:11
$
```

将 who2 与系统的 who 对比一下，除了某些字段的宽度有些不同外，其他的都完全一致，而要调整宽度也是很容易的。

这里的 who 只列出了 3 个字段：用户名、终端类型和登录时间，有些版本的 who 还会列出用户所在主机的信息，这个功能在 who2 中通过预编译选项 SHOWHOST 控制。

2.5.6 回顾与展望

这一章是从这样一个简单的问题开始的：Unix 的 who 命令是如何工作的？接下来分 3 步走，首先弄清 who 的功能，然后通过联机帮助和头文件知道了 who 的工作原理，最后，通过编写自己的 who 来检验对知识的掌握程度。

在这 3 步中，学习了如何使用联机帮助，如何使用头文件，了解了记录登录信息的 utmp 文件的结构，知道了 Unix 处理时间的方式，学会了通过相关主题来获取信息，这些知识对掌握 Unix 编程是很重要的。通过编写程序，更加深化了对知识的理解和掌握。

2.6 编写 cp(读和写)

在 who 命令中介绍了如何读文件，接下来要通过 cp 命令来学习如何写文件。

2.6.1 问题 1：cp 命令能做些什么

cp 能够复制文件，典型的用法是：

```
$ cp source-file target-file
```

如果 target-file 所指定的文件不存在，cp 就创建这个文件，如果已经存在就覆盖，target-file 的内容与 source-file 相同。

2.6.2 问题 2：cp 命令是如何创建/重写文件的

1. 创建/重写文件

创建或重写文件的一种方法是使用系统调用函数 creat，creat 的用法如下：

creat		
目标	创建/重写一个文件	
头文件	#include <fcntl.h>	
函数原型	int fd = creat(char *filename, mode_t mode)	
参数	filename	文件名
	mode	访问模式
返回值	-1	遇到错误
	fd	成功创建

creat 告诉内核创建一个名为 filename 的文件，如果这个文件不存在，就创建它，如果已经存在，就把它的内容清空，把文件的长度设为 0。

如果内核成功地创建了文件，那么文件的许可位（permission bits）被设置为由第 2 个参数 mode 所指定的值，如：

```
fd = creat("addressbook",0644);
```

创建一个名为 addressbook 的文件，如果文件不存在，那么文件的许可位被设为 rw-r-r--（参见第 3 章）。如果文件已经存在，它的内容会被清空。任一种情况下，fd 都会是指向 addressbook 的文件描述符。

2. 写文件

用 write 系统调用向已打开的文件中写入数据。

write		
目标	将内存中的数据写入文件	
头文件	#include <unistd.h>	
函数原型	ssize_t result = write(int fd, void * buf, size_t amt)	
参数	fd	文件描述符
	buf	内存数据
	amt	要写的字节数
返回值	-1	遇到错误
	num written	成功写入

write 这个系统调用告诉内核将内存中指定的数据写入文件，如果内核不能写入或写入失败，write 返回-1，如果写入成功，则返回写入的字节数。

为什么实际写入的字节数会少于所要求的呢？有两个原因，第一个是有的系统对文件的最大尺寸有限制，第二个是磁盘空间接近满了。在上述两种情况下，内核都会尽量把数据往文件中写，并将实际写入的字节数返回，所以调用 write 后都必须检查返回值是否与要写入的相同，如果不同，就要采取相应的措施。

2.6.3 问题 3：如何编写 cp

下面通过编写一个实际的 cp 来检查对知识的理解，程序的流程如下：

```
    open sourcefile for reading
    open copyfile for writing
+->read from source to buffer  --  eof?   -+
|__  write from buffer to copy             |
                                           |
    close sourcefile        <--------------+
    close copyfile
```

图 2.4 显示了涉及的对象及数据流的走向。

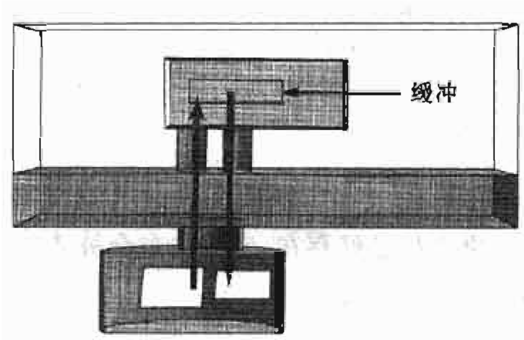


图 2.4　通过读和写来复制文件

文件在磁盘上，源文件在左边，右边的是目标文件。进程在用户空间，缓冲区是进程内存的一部分，进程有两个文件描述符，一个指向源文件，一个指向目标文件，从源文件中读取数据写入缓冲，再将缓冲中的数据写入目标文件。下面就是实现上述逻辑的代码：

```
/** cp1.c
 *  version 1 of cp - uses read and write with tunable buffer size
 *
 *  usage: cp1 src dest
 */
#include    <stdio.h>
#include    <unistd.h>
#include    <fcntl.h>

#define BUFFERSIZE   4096
#define COPYMODE     0644

void oops(char *, char *);

main(int ac, char *av[])
{
    int     in_fd, out_fd, n_chars;
    char    buf[BUFFERSIZE];
                                                        /* check args */
    if ( ac != 3 ){
        fprintf( stderr, "usage: %s source destination\n", *av);
        exit(1);
    }
                                                        /* open files */

    if ( (in_fd=open(av[1], O_RDONLY)) == -1 )
        oops("Cannot open ", av[1]);

    if ( (out_fd=creat( av[2], COPYMODE)) == -1 )
        oops( "Cannot creat", av[2]);
                                                        /* copy files */
```

```
    while ( (n_chars = read(in_fd , buf, BUFFERSIZE)) > 0 )
        if ( write( out_fd, buf, n_chars ) != n_chars )
            oops("Write error to ", av[2]);
    if ( n_chars == -1 )
        oops("Read error from ", av[1]);
                                                /* close files */

    if ( close(in_fd) == -1 || close(out_fd) == -1 )
        oops("Error closing files","");
}

void oops(char * s1, char * s2)
{
    fprintf(stderr,"Error: %s ", s1);
    perror(s2);
    exit(1);
}
```

编译并测试上述代码：

```
$ cc cp1.c -o cp1
$ cp1 cp1 copy.of.cp1
$ ls -l cp1 copy.of.cp1
-rw-r--r--   1 bruce  bruce 37419 Jul 23 03:12 copy.of.cp1
-rwxrwxr-x   1 bruce  bruce 37419 Jul 23 03:08 cp1
$ cmp cp1 copy.of.cp1
$
```

用 Unix 所带的文件比较工具 cmp 对上述两个文件的内容做比较。cmp 没有给出任何提示，说明它们的内容完全相同。

接下来看看程序对错误输入的反映，在命令行输入：

```
$ cp1 xxx123 file1
Error: Cannot open xxx123: No such file or directory
$ cp1 cp1 /tmp
Error: Cannot creat /tmp: Is a directory
```

阅读与系统调用相关的联机帮助，看看还有什么错误可能发生，逐一地测试这些错误情况。注意不要覆盖掉那些想要保存的文件。

2.6.4　Unix 编程看起来好像很简单

who 命令从文件中读数据然后以一定的格式输出，cp 命令从一个文件中读数据然后写入到另外的文件中，它们用到的是类似的系统调用，都是在内存和文件之间交换数据，可以从联机帮助和头文件中得到足够的信息来进行编程。

这样看来，Unix 编程好像真的不难，是不是还有些东西没涉及到？答案是肯定的，还记

不记得那 3 个问题？在这里要多问一个问题：如何使你的程序运行得更加有效？

2.7　提高文件 I/O 效率的方法：使用缓冲

cp1 中定义了 BUFFERSIZE 这个常量，用于标识每次读/写操作的数据长度，这里的值是 4096，接下来是个很重要的问题：缓冲区的大小对性能有影响吗？

2.7.1　缓冲区的大小对性能的影响

缓冲区的大小对性能有很大的影响，举例来说，用勺子把汤从一个碗里舀到另一个碗里，用较大的勺子就可以少舀几次，从而节省时间。

对文件操作而言也是这样的，来看对一个 2500 字节的文件的 copy 操作：

```
文件大小 = 2500 字节
如果缓冲区大小 = 100 字节
    那么需要 25 次 read()和 25 次 write()
如果 缓冲区大小 = 1000 字节
    那么需要 3 次 read()和 3 次 write()
```

把缓冲区从 100 字节增加到 1000 字节会使系统调用的次数从 50 次减少到 6 次，这确实很可观。

复制一个 5MB 大小的文件，不同的缓冲区所对应的执行时间如下：

缓冲大小	执行时间/s
1	50.29
4	12.81
16	3.28
32	0.96
128	0.56
256	0.37
512	0.27
1024	0.22
2048	0.19
4096	0.18
8192	0.18
16384	0.18

系统调用是需要时间的，程序中频繁的系统调用会降低程序的运行效率。

2.7.2　为什么系统调用需要很多时间

为什么系统调用会消耗很多时间？参见图 2.5 所示的控制流程。

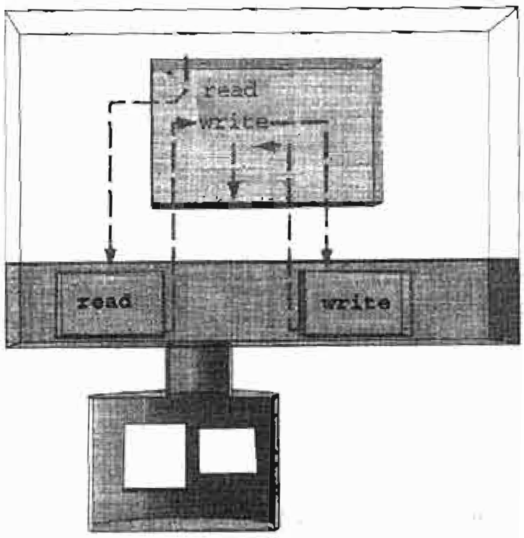


图 2.5　系统调用时的控制流图

图 2.5 中用户进程位于用户空间，内核位于系统空间，磁盘只能被内核直接访问。程序 cp1 要读取磁盘上的数据只能通过系统调用 read，而 read 的代码在内核中，所以当 read 调用发生时，执行权会从用户代码转移到内核代码，执行内核代码是需要时间的。

系统调用的开销大不仅仅是因为要传输数据，当运行内核代码时，CPU 工作在管理员（supervisor，又称超级用户）模式，这对应于一些特殊的堆栈和内存环境，必须在系统调用发生时建立好。系统调用结束后（read 返回时），CPU 要切换到用户模式，必须把堆栈和内存环境恢复成用户程序运行时的状态，这种运行环境的切换要消耗很多时间。

当工作在管理员模式下，程序可以直接访问磁盘、终端、打印机等设备，还可以访问全部的内存空间，而在用户模式，程序不能直接访问设备，也只能访问特定部分的内存空间。在运行时刻，系统会根据需要不断地在两种模式间切换。管理员模式和用户模式的切换与 CPU 关系很大，CPU 中有特定的标记来区分当前的工作模式，而 Unix 系统的设计必须考虑到 CPU 的这种特点，才能够实现不同工作模式间的良好切换。

举个影片超人的例子，当肯特（生活中的超人）要从用户模式（普通人）切换到管理员模式（超人）时，他得先找个地方，比如电话亭，脱下西装，摘掉眼镜，再改变发型，变成超人后才能去拯救别人，事情完了以后，还得找个地方变回普通人。变来变去是需要时间的，要是肯特整天忙于变来变去，就不会有太多的时间来拯救人类了。

在计算机的世界中也是一样，要是 CPU 把太多的时间消耗在执行内核代码和模式切换上，就不可能有很多时间来执行程序中业务逻辑的代码或提供系统服务，所以要尽可能地减少模式间的切换。对系统来说这种时间上的开销是昂贵的，那么 who 版本花费在读、写数据的时间开销有多大呢？

2.7.3　低效率的 who2.c

每次从 utmp 中读出一条记录，就如同要煎 3 个荷包蛋，每次到超市去买一个鸡蛋，煎好

了再去买一个，这是很低效率的方法，完全可以一次把 3 个鸡蛋都买回来。对于 who 而言，可以一次读入多个记录放在缓冲区中，下面是实现上述想法的伪代码：

```
getegg(){
    if ( eggs_left_in_carton == 0 )
        refill carton at store
        if ( eggs_at_store == 0 )
            return EndOfEggs
    }
    eggs_left_in_carton -- ;
    return one egg;
}
```

getegg 的每次调用会从篮子里拿一个鸡蛋，而不是每次都要到超市去买，只有当篮子空了才需要去超市。

参见/usr/include/stdio. h 中 getc 的代码，getc 的实现使用了与 getegg 类似的算法。

2.7.4 在 who2. c 中运用缓冲技术

在 who2. c 中加入缓冲机制可以提高程序的运行效率，接下来要把 getegg 中的想法用代码实现，如图 2.6 所示。

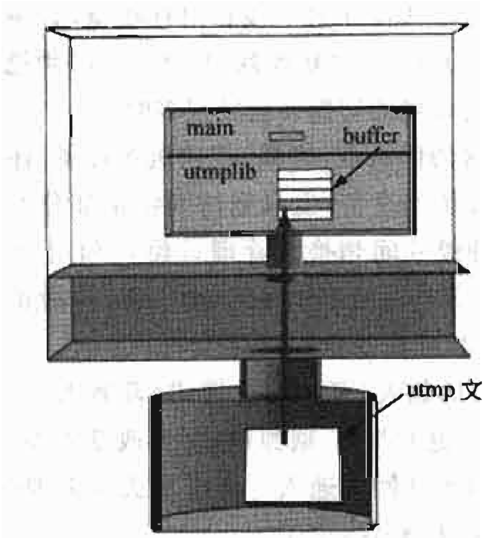


图 2.6 在用户空间数据加入磁盘缓冲

用一个能容纳 16 个 utmp 结构的数组作为缓冲区，在图 2.6 中标识为 buffer，就像你一次会买很多个鸡蛋一样，buffer 可以存放很多数据。编写 utmp_next 函数来从缓冲区中取得下一个 utmp 结构的数据。

修改原来的主函数 main，通过调用 utmp_next 来取得数据，当缓冲区的数据都被取出后，utmp_next 会调用 read，通过内核再次获得 16 条记录充满缓冲区。用这种方法可以使

read 的调用次数减少到原来的 1/16。

以上算法在 utmplib.c 中加以实现。

```
/* utmplib.c  - functions to buffer reads from utmp file
 *
 *      functions are
 *          utmp_open( filename )     - open file
 *              returns -1 on error
 *          utmp_next( )              - return pointer to next struct
 *              returns NULL on eof
 *          utmp_close()              - close file
 *
 *      reads NRECS per read and then doles them out from the buffer
 */
#include    <stdio.h>
#include    <fcntl.h>
#include    <sys/types.h>
#include    <utmp.h>

#define NRECS 16
#define NULLUT ((struct utmp *)NULL)
#define UTSIZE (sizeof(struct utmp))

static char   utmpbuf[NRECS * UTSIZE];              /* storage */
static int    num_recs;                              /* num stored */
static int    cur_rec;                               /* next to go */
static int    fd_utmp = -1;                          /* read from */

utmp_open( char * filename )
{
    fd_utmp = open( filename, O_RDONLY );            /* open it */
    cur_rec = num_recs = 0;                          /* no recs yet */
    return fd_utmp;                                  /* report */
}

struct utmp * utmp_next()
{
    struct utmp * recp;
    if ( fd_utmp == -1 )                             /* error ? */
       return NULLUT;
    if ( cur_rec == num_recs && utmp_reload() == 0 ) /* any more ? */
       return NULLUT;
                                                     /* get address of next record */
    recp = ( struct utmp * ) &utmpbuf[cur_rec * UTSIZE];
    cur_rec++;
    return recp;
```

```
}

int utmp_reload()
/*
 * read next bunch of records into buffer
 */
{
    int amt_read;                                          /* read them in */
    amt_read = read( fd_utmp , utmpbuf, NRECS * UTSIZE );  /* how many did we get? */
    num_recs = amt_read/UTSIZE;                            /* reset pointer */
    cur_rec = 0;
        return num_recs;
}

utmp_close()
{
    if ( fd_utmp != -1 )                                   /* don't close if not */
        close( fd_utmp );                                  /* open */
}
```

utmplib.c 包含使用缓冲区所需的变量和函数，变量 num_recs 记录了缓冲区中的数据个数，变量 cur_rec 记录了缓冲区中已被使用的数据的个数。

每次要从缓冲区中读数据前，先检查 cur_rec 的值是否等于 num_recs，如果相等说明缓冲区中已经没有可用数据了，就调用 read 从硬盘上读数据来填满缓冲区，在返回数据前，增加 cur_rec 的值。

函数 utmp_next 返回指向结构的指针，utmplib.c 隐藏了实现细节，提供简单清晰的操作缓冲区的接口。

下面是修改后的 main：

```
/* who3.c - who with buffered reads
 *    - surpresses empty records
 *    - formats time nicely
 *    - buffers input (using utmplib)
 */
#include    <stdio.h>
#include    <sys/types.h>
#include    <utmp.h>
#include    <fcntl.h>
#include    <time.h>

#define SHOWHOST

void show_info(struct utmp * );
void showtime(time_t);
```

```
int main()
{
    struct utmp  * utbufp, /* holds pointer to next rec */
            * utmp_next(); /* returns pointer to next */

    if ( utmp_open( UTMP_FILE ) == -1 ){
       perror(UTMP_FILE);
       exit(1);
    }
    while ( ( utbufp = utmp_next() ) != ((struct utmp * ) NULL) )
        show_info( utbufp );
    utmp_close( );
    return 0;
}
/*
 * show info()
...
```

修改后的主函数没有直接对 open、read 和 close 进行调用，而是调用与之等价的具有缓冲模式的函数接口。用于显示的函数 show_info 没有受到任何影响。

2.8 内核缓冲技术

应用缓冲技术对提高系统的效率是很明显的，它的主要思想是一次读入大量的数据放入缓冲区，需要的时候从缓冲区取得数据。

内核使用缓冲吗

管理员模式和用户模式之间的切换需要消耗时间，相比之下，磁盘的 I/O 操作消耗的时间更多，为了提高效率，内核也使用缓冲技术来提高对磁盘的访问速度，如图 2.7 所示。

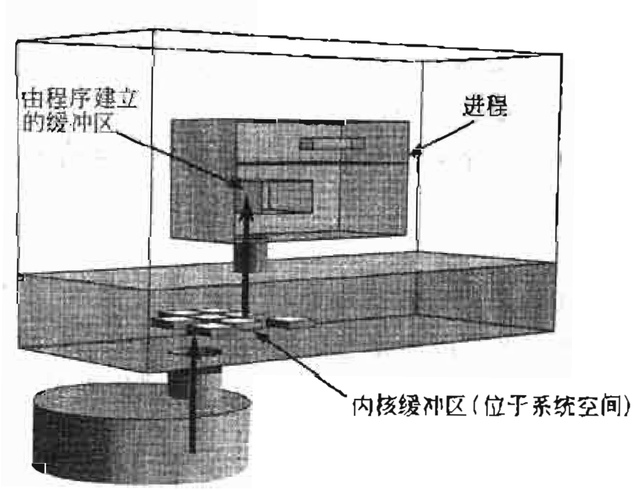


图 2.7 内核缓冲磁盘上的数据

正如 utmp 文件是用户登录记录的集合，磁盘是数据块的集合，内核会对磁盘上的数据块作缓冲，就像 who 程序缓冲 utmp 记录一样。内核将磁盘上的数据块复制到内核缓冲区中，当一个用户空间中的进程要从磁盘上读数据时，内核一般不直接读磁盘，而是将内核缓冲区中的数据复制到进程的缓冲区中。

当进程所要求的数据块不在内核缓冲区时，内核会把相应的数据块加入到请求数据列表中，然后把该进程挂起，接着为其他进程服务。

一段时间之后(很短)，内核把相应的数据块从磁盘读到内核缓冲区，然后再把数据复制到进程的缓冲区中，最后唤醒被挂起的进程。

理解内核缓冲技术的原理有助于更好地掌握系统调用 read 和 write，read 把数据从内核缓冲区复制到进程缓冲区，write 把数据从进程缓冲区复制到内核缓冲区，它们并不等价于数据在内核缓冲和磁盘之间的交换。

从理论上讲，内核可以在任何时候写磁盘，但并不是所有的 write 操作都会导致内核的写动作。内核会把要写的数据暂时存在缓冲区中，积累到一定数量后再一次写入。有时会导致意外情况，比如突然断电，内核还来不及把内核缓冲区中的数据写到磁盘上，这些更新的数据就会丢失。

应用内核缓冲技术导致的结果：

- 提高磁盘 I/O 效率
- 优化磁盘的写操作
- 需要及时地将缓冲数据写入磁盘

2.9 文件读写

who 是从文件读数据，cp 从一个文件读数据写入到另一个文件中，会不会有对同一个文件既读又写的情况呢？

2.9.1 注销过程：做了些什么

在注销过程中，系统改变了文件 utmp 中相应的登录记录，从 who1 的输出可以看出文件 utmp 中还包含那些未被使用的终端的信息，通过实验来看这个过程：

1. 分别从两个窗口登录到一个 Unix 系统中。
2. 运行前面编写的 who1 程序来察看 utmp 中与你的登录相关的两条记录。注意用来登录的是哪一个终端。
3. 注销其中一个。
4. 运行 who1 察看上述两条记录内容的变化。

你会发现其中一条的用户名字段跟原来的不同，它的 ut_time 字段的值也发生了改变。在有些系统中，用户名是被清空的。还请关心该记录还有哪些改变？如果是远程登录呢？

2.9.2 注销过程：如何工作的

这其实很简单，要把用户名清空，按以下步骤做就行了：

1. 打开文件 utmp;
2. 从 utmp 中找到包含你所在终端的登录记录;
3. 对当前记录做修改;
4. 关闭文件。

下面详细讨论这 4 个步骤。

1. *打开文件 utmp*

因为负责注销的程序必须对文件 utmp 进行读(找到登录记录)和写(修改登录记录)操作,所以必须先打开这个文件:

```
fd = open(UTMP_FILE, O_RDWR);
```

2. *从 utmp 中找到包含你所在终端的登录记录*

这一步很简单,在 while 循环中读取一条 utmp 记录,将它的 ut_line 字段跟你的终端的名字做比较,如果相等则调用修改函数:

```
while ( read(fd, rec, utmplen) == utmplen)      /* get next record */
   if ( strcmp(rec.ut_line, myline) == 0 )      /* what, my line? */
      revise_entry();                           /* remove my name */
```

3. *对当前记录做修改*

负责注销的程序修改当前记录,再把它写回到文件 utmp 中。具体来说,要把 ut_type 的值从 USER_PROCESS 改成 DEAD_PROCESS,把 ut_time 字段的值改为注销时间,也就是当前时间,有些版本会把用户名和远程主机字段的内容清空,这些代码编写起来很容易。

接下来要处理一个棘手的问题,如何把修改过的记录写回文件? 可以用 write 吗? 不行,write 只会更新下一条记录,而不是当前那条要修改的记录。因为系统每次打开一个文件都会保存一个指向文件当前位置的指针,当读写操作完成时,指针会移到下一个记录位置,这个指针与文件描述符相关联。在这种情况下,指针是指向下一条登录记录的头一个字节,这引出了一个重要的问题:

问题:在文件操作中,如何改变一个文件的当前读/写位置?

答题:使用系统调用 lseek。

4. *关闭文件*

调用 close(fd)。

2.9.3　改变文件的当前位置

前面讲过 Unix 每次打开一个文件都会保存一个指针来记录文件的当前位置,如图 2.8 所示。

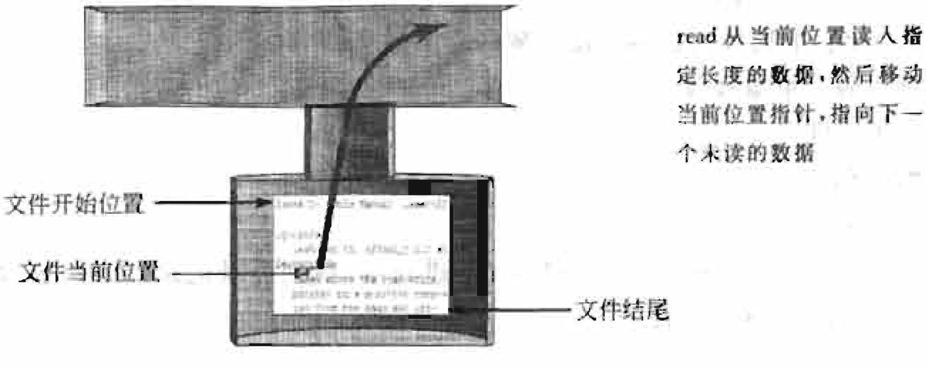


图 2.8　内核为每个打开的文件保存一个位置指针

当从文件读数据时，内核从指针所标明的地方开始，读取指定的字节，然后移动位置指针，指向下一个未被读取的字节，写文件的操作也是类似的。

指针是与文件描述符相关联的，而不是与文件关联，所以如果两个程序同时打开一个文件，这时会有两个指针，两个程序对文件的读操作不会互相干扰。

系统调用 lseek 可以改变已打开文件的当前位置，lseek 的用法如下：

lseek		
目标	使指针指向文件中的指定位置	
头文件	#include <sys/type.h> #include <unistd.h>	
函数原型	off_t oldpos = lseek(int fd, off_t dist, int base)	
参数	fd	文件描述符
	dist	移动的距离
	base	SEEK_SET => 文件的开始 SEEK_CUR => 当前位置 SEEK_END => 文件结尾
返回值	-1	遇到错误
	oldpos	指针变化前的位置

lseek 改变文件描述符所关联的指针的位置，新的位置由 dist 和 base 来指定，base 是基准位置，dist 是从基准位置开始的偏移量。基准位置可以是文件的开始(0)、当前位置(1)或文件的结尾(2)。如：

```
lseek(fd, -(sizeof(struct utmp)), SEEK_CUR);
```

把指针往前移一个 utmp 结构，注意偏移量可以是负的。

```
lseek(fd, 10 * sizeof(struct utmp), SEEK_SET);
```

上述代码把指针指到第 11 个记录的开始位置。下面的代码：

```
lseek(fd, 0, SEEK_END);
```

```
write(fd, "hello", strlen("hello"));
```

使指针指到文件的末尾，接下来写一个字符串到文件中。

最后要说明的是，lseek(fd，0，SEEK_CUR)返回指针所指向的当前位置。

2.9.4 编写终端注销的代码

利用上述知识就可以编写一个函数对注销的的用户修改 utmp 中相应的记录：

```
/*
 * logout_tty(char * line)
 * marks a utmp record as logged out
 * does not blank username or remote host
 * return - 1 on error, 0 on success
 */
int logout_tty(char * line)
{
   int          fd;
   struct utmp  rec;
   int          len = sizeof(struct utmp);
   int          retval = - 1;                      /* pessimism */

   if (( fd = open(UTMP_FILE,O_RDWR)) == - 1 )     /* open file */
      return - 1;
   /* search and replace */
   while (read(fd, &rec, len) == len )
      if (strncmp(rec.ut_line, line, sizeof(rec.ut_line)) == 0)
      {
         rec.ut_type = DEAD_PROCESS;               /* set type */
         if (time(&rec.ut_time != - 1))            /* and time */
            if ( lseek(fd, - len, SEEK_CUR) != - 1) /* back up */
               if (write(fd, &rec, len) == len)    /* update */
                  retval = 0;                      /* success! */
         break;
      }
   /* close the file */
   if (close(fd) == - 1)
      retval = - 1;
   return retval;
}
```

上述这段代码对每个系统调用都有出错处理，这是很有必要的，因为这个程序需要修改一些重要的系统配置文件，如果这些文件的内容遭到破坏，那系统就会遇到麻烦。实际上，这本书中不是所有的地方都有很完善的出错处理，这并不表示出错处理不重要，而是为了使程序更加清晰易懂。

接下来看一看出错处理与报告。

2.10　处理系统调用中的错误

如果 open 无法打开指定的文件，它会返回－1。同样地，当 read 无法读的时候，它会返回－1，当 lseek 无法指定指针位置时，它也会返回－1，－1 是表示在系统调用中出了些问题，调用者每次都必须检查返回值，一旦检测到错误，必须做出相应的处理。

系统调用会遇到哪些错误呢？每个系统调用都有自己的错误集，以 open 为例，如果要打开的文件不存在，或者虽然存在，但没有读的权限，或者已经打开的文件太多，都会导致系统报错。如何来确定发生了哪一种错误呢？

1. 确定错误的种类：errno

内核通过全局变量 errno 来指明错误的类型，每个程序都可以访问到这个变量。

在 error(3)的联机帮助和<errno.h>中包含错误代码和相应的说明，以下是一些例子：

```
#define  EPERM   1   /* Operation not permitted */
#define  ENOENT  2   /* No such file or directory */
#define  ESRCH   3   /* No such process */
#define  EINTR   4   /* Interrupted system call */
#define  EIO     5   /* I/O error */
```

2. 不同的错误需要不同的处理

根据以上列出的错误类型，在程序中进行相应的处理：

```
#include <errno.h>
extern int errno;
int sample()
{
    int fd;
    fd = open("file", O_RDONLY);
    if (fd == -1)
    {
        printf("Cannot open file: ");
        if ( errno == ENOENT )
            pritnf("There is no such file.");
        else if ( errno == EINTR )
            printf("Interrupted while opening file.");
        else if ( errno == EACCESS )
            printf("You do not have permission to open file.");
        ...
```

需要根据不同的错误类型做不同的错误处理。如果要打开的文件不存在，那么给出提示重新输入文件名，如果已经打开的文件太多，那就关闭一些不需要的文件，这种情况是不需要给用户任何提示的。

3. 显示错误信息：perror(3)

在需要显示出错误信息的时候，可以根据不同的错误代码打印相应的字符串，上面的例子 sample()就是这样做的，另外一种更简便的方法是用 perror(string)这个函数，它会自己查找错误代码，在标准错误输出中显示出相应的错误信息，参数 string 是要同时显示出的描述性信息。

应用了 perror 的 sample：

```
int sample()
{
    int fd;
    fd = open("file", O_RDONLY);
    if (fd == -1)
    {
        perror("Cannot open file");
        return;
    }
    ...
```

当有错误发生时，可能会看到如下的信息：

```
Cannot open file: No such file or directory
Cannot open file: Interrupted system call
```

显示的第一部分是用户传递进去的描述性信息，第二部分是根据错误代码查到的错误提示。

小　　结

1. 主要内容

- who 命令通过读系统日志的内容显示当前已经登录的用户。
- Unix 系统把数据存放在文件中，可以通过以下系统调用操作文件：

```
open(filename, how)
creat(filename, mode)
read(fd, buffer, amt)
write(fd, buffer, amt)
lseek(fd, distance, base)
close(fd)
```

- 进程对文件的读/写都要通过文件描述符，文件描述符表示文件和进程之间的连接。
- 每次系统调用都会导致用户模式和内核模式的切换以及执行内核代码，所以减少程序中的系统调用发生的次数可以提高程序的运行效率。
- 程序可以通过缓冲技术来减少系统调用的次数，仅当写缓冲区满或读缓冲区空时才调用内核服务。

- Unix 内核可以通过内核缓冲来减少访问磁盘 I/O 的次数。
- Unix 中时间的处理方式是记录从某一个时间开始经过的秒数。
- 当系统调用出错时会把全局变量 errno 的值设为相应的错误代码，然后返回－1，程序可以通过检查 errno 来确定错误的类型，并采取相应的措施。
- 这一章涉及的知识在系统中都可以找到，联机帮助中有命令的说明，有些还会涉及命令的实现，头文件中有结构和系统常量的定义，还有函数原型的说明。

2. *习题*

2.1　在 Unix 中有一个 w 命令，这个命令与 who 有关，运行这个命令，并阅读它的联机帮助，找出它提供了哪些 who 没有提供的信息？其中有哪些信息来自 utmp 这个文件？这些信息各是什么含义？其他信息来自哪里？

2.2　用户登录的时候，登录信息会被记录在 utmp 文件中，在注销的时候，文件中相应的记录会被删除。如果用户正在使用系统的时候，系统突然崩溃，很明显，这时候 utmp 文件的内容会有问题，在系统重新启动的时候，会对 utmp 做什么处理呢？系统会不会重新为每一条可用终端建立记录？查阅相关的联机帮助和头文件，以及启动脚本，来找出上述问题的答案，可以在自己的机器上做实验来验证自己的想法。

2.3　做个实验，把一个文件复制到/dev/tty：cp1 cp1.c /dev/tty。这时复制的目标文件是一个终端。对终端的读写操作与对一个普通文件进行读写是一样的。接下来做实验，从终端读，这时会从键盘读字符，然后写入到文件中，输入是以回车 ＋ <Ctrl-D>作为结束标志的。

2.4　标准 C 函数如 fopen、getc、fclose、fgets 的实现都包含内核级的缓冲，它们用到了一个结构 FILE，并以此为基础构造了类似 utmplib 的中间层，在头文件中找到结构 FILE 的成员描述，将其与 utmplib.c 中的变量做比较。

2.5　怎么来确定数据已经被写到磁盘上(不是被缓冲)？采用缓冲技术时，内核会把要写入磁盘的数据放在缓冲区，然后在它认为合适的时候写入磁盘。阅读相应的联机帮助，找到能够确定地把数据写入磁盘的方法。

2.6　Unix 允许一个文件同时被多个进程打开，也允许一个进程同时打开好几个文件，做多次打开文件的实验：

(1) 以读的方式打开文件

(2) 以写的方式打开文件

(3) 再次以读的方式打开文件

这时有 3 个文件描述符，接下来：

(4) 从第一个文件描述符(以下简称 fd)中读 20 字节，显示读到的内容。

(5) 从第二个 fd 写入“testing 123…”。

(6) 从第三个 fd 读出 20 字节，显示读到的内容。

2.7　联机帮助 man 中可以查到关于命令、系统调用、系统设备等帮助信息，如何才能了解 man 的使用方法？在你的系统中，man 分为几个小节？它们分别是什么？

2.8　本章提到文件 utmp 中还包含了一些记录，它们与已登录用户的信息无关，那么它们存放的是什么？各代表什么含义？

2.9　lseek 可以将文件指针移动文件的末尾以后，如：

```
lseek(fd, 100, SEEK_END)
```

将指针移到文件末尾再往后 100 个字节的地方。如果从文件末尾以后 100 个字节的地方开始读会出现什么情况？如果从文件末尾以后 100 个字节的地方开始写会出现什么情况？从 100 字节增加到 20000 字节，再写入“hello”看看会有什么结果，用 ls -l 来检查文件的长度，再用 ls -s 试试。

3. 编程练习

2.10　从 who 的联机帮助中可以知道 who am i 也是可以接受的形式，同样还有 whoami，修改 who2.c，使它支持 who am i 的形式。阅读 whoami 的联机帮助，看它与 who 有什么不同，编程实现 whoami。

2.11　使用标准 cp 命令时，当原文件和目标文件相同时，会有什么结果？修改 who2.c 使之能够处理这种情况。

2.12　在 utmplib.c 中的几个函数是为了提高 utmp 文件的读写效率，调用这些函数，每次返回一个 utmp 记录，有时返回的记录中可能不包含任何有用的信息，修改 utmplib.c，使每次返回的都是有用的信息。这样做会影响到 who3.c 其他部分的代码吗？为什么？

2.13　函数 logout_tty()使用 lseek 往前移动指针，以便重写当前记录，注意在这个函数中没有用到缓冲技术，如果使用缓冲可以提高程序的运行效率。

(1) 如果在 logout_tty()中用到 utmplib.c 中的函数会产生什么问题？

(2) 在 utmplib.c 中增加一个函数：

```
utmp_seek(record_offset, base)
```

改变当前指针的位置，record_offset 是要移动的偏移量，base 可以是 SEEK_SET、CEEK_CUR 或 SEEK_END，注意偏移量每增加 1，表示要移动 sizeof(struct utmp)的字节。

(3) 修改 logout_tty()以便能够使用 utmplib.c 中的函数。

2.14　who1 可以显示出 utmp 中的每条记录，虽然这不是原来想要的功能，但却提供了一个工具，以便能够检查 utmp 文件内容的变化。utmp 记录中的 ut_type 字段很有用，修改 who1 使之能够显示 utmp 记录的所有字段，进一步修改使 who1 能够通过参数指定要读取的文件，这样就可以用 who1 来检查文件 wtmp 的内容。

2.15　标准的 cp 会自动覆盖已经存在的文件，而不给出任何提示，如已经存在了一个文

件 file2，又输入：

```
$ cp file1 file2
```

会覆盖 file2 的内容。标准的 cp 有一个参数 -i 可以在覆盖前给出提示，得到确认后才覆盖。修改 cp1.c 使之也具有上述特性。

4. 项目

根据本章所学的内容，编写以下的 Unix 程序：

```
ac, last, cat, head, tail, od, dd
```

5. 最后一个问题：tail 命令

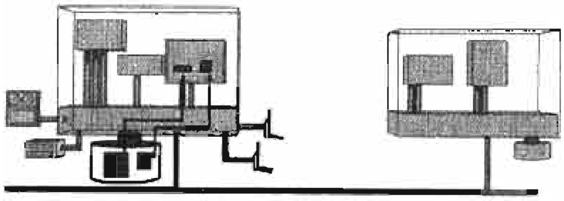
这一章通过分析几个命令已经学到了很多知识，最后还有一个命令 tail，请读者来分析。lseek 命令可以移动指针，lseek(fd, 0, SEEK_END)可以使指针移到文件的末尾。

tail 命令显示出文件末尾指定行数的内容，所以，tail 必须能够找到文件中一个指定的地方，注意不是末尾。

这如何来实现呢？开动脑筋好好想想，注意用缓冲技术来提高效率。阅读联机帮助看看 tail 都有哪些选项，想想它们怎么实现。

本书的网站上有两个版本的 tail 的源代码，其中一个是 GNU 版本的，另外一个是 BSD 版本的，它们用了完全不同的思路，却都实现得很好，最好先自己写，然后再参考它们的实现。

第 3 章　目录与文件属性：编写 ls



概念与技巧

- 目录是文件的列表
- 如何读取目录的内容
- 文件类型以及如何知道文件的类型
- 文件属性以及如何知道文件的属性
- 位操作及掩码的使用
- 用户与组 ID 及 passwd 数据库

相关系统调用与函数

- opendir、readdir、closedir、seekdir
- stat
- chmod、chown、utime
- rename

相关命令

- ls

3.1 介　　绍

已经介绍了如何读/写文件内容的方法。除了内容之外，文件还有很多属性，比如文件所有者、最后修改时间、文件大小、类型等。文件名在目录中列出，正如电话号码簿中列有人名一样。如何读取文件名和文件的属性呢？

ls 命令可以列出目录中所有文件的名字，以及这些文件的其他信息。本章通过分析 ls 命令来学习目录和文件的类型与属性。

3.2 问题 1：ls 命令能做什么

3.2.1 ls 可以列出文件名和文件的属性

在命令行输入 ls：

```
$ ls
Makefile  docs    ls2.c    s.tar     statdemo.c  tail1.c
chap03    ls1.c  old_src  stat1.c   tail1
$
```

ls 的默认动作是找出当前目录中所有文件的文件名，按字典序排序后输出。有些版本的 ls 默认会分栏输出，有些需要参数 -C 才这样做。

ls 还能显示其他的信息，如果加上 -l 选项，ls 会列出每个文件的详细信息，也叫 ls 的长格式：

```
$ ls -l
total 108
-rw-rw-r--  2 bruce users      345 Jul 29 11:05 Makefile
-rw-rw-r--  1 bruce users    27521 Aug  1 12:14 chap03
drwxrwxr-x  2 bruce users     1024 Aug  1 12:15 docs
-rw-r--r--  1 bruce users      723 Feb  9  1998 ls1.c
-rw-r--r--  1 bruce users     3045 Feb 15 03:51 ls2.c
drwxrwxr-x  2 bruce users     1024 Aug  1 12:14 old_src
-rw-rw-r--  1 bruce users    30720 Aug  1 12:05 s.tar
-rw-r--r--  1 bruce support    946 Feb 18 17:15 stat1.c
-rw-r--r--  1 bruce support    191 Feb  9  1998 statdemo.c
-rwxrwxr-x  1 bruce users    37351 Aug  1 12:13 tail1
-rw-r--r--  1 bruce users     1416 Aug  1 12:05 tail1.c
$
```

每一行代表一个文件和它的多个属性。

3.2.2　列出指定目录或文件的信息

一个 Unix 系统中会有很多的目录，每个目录中又会有很多文件。如果要列出一个非当前目录的内容或者是一个特定文件的信息，则需要在参数中给出目录名或文件名。

用 ls 列出指定目录包含的文件

例　子	说　明
ls /tmp	列出/tmp 目录中各文件的文件名
ls -l docs	列出 docs 目录中各文件的属性
ls -l ../Makefile	显示文件../Makefile 的属性
ls *.c	显示与 *.c 匹配的文件

如果参数是目录，ls 列出目录的内容，如果参数是文件，ls 列出文件名和属性。所给的命令行选项在很大程度上决定了 ls 的输出内容。

3.2.3 经常用到的命令行选项

命 令	说 明
ls -a	列出的内容包含以"."开头的文件
ls -lu	显示最后访问时间
ls -s	显示以块为单位的文件大小
ls -t	输出时按时间排序
ls -F	显示文件类型

如果对 Unix 不是很熟悉，那么可能需要解释一下 -a 这个选项，在 Unix 中，ls 一般不会列出以"."开始的文件，所以可以把这样的文件看作是隐藏文件。ls 加上了 -a 以后，遇到这样的文件也必须把它列出来。对操作系统(例如内核)而言，文件名前面的"."没有任何特殊的含义，它只对 ls 的使用者有意义。

某些应用程序的配置文件是位于用户的主目录下以"."开始的某个文件，这是由习惯形成的，因为在大多数情况下可以将它们隐藏。但是需要时可以直接被打开编辑，不需要任何特殊的操作。

3.2.4 问题 1 的答案

通过实验和联机帮助可以知道 ls 做了以下两件事：

- 列出目录的内容
- 显示文件的信息

注意 ls 对文件和目录所做的操作是不同的。ls 能判定参数指定的是文件还是目录。这是如何做到的呢？如果要自己写一个 ls，以下三点是需要掌握的：

- 如何列出目录的内容
- 如何读取并显示文件的属性
- 给出一个名字，如何能够判断出它是目录还是文件

3.3 文 件 树

在开始之前，先来看看 Unix 是如何组织磁盘上的文件的。

磁盘上的文件和目录被组成一棵目录树，每个节点都是目录或文件，如图 3.1 所示。

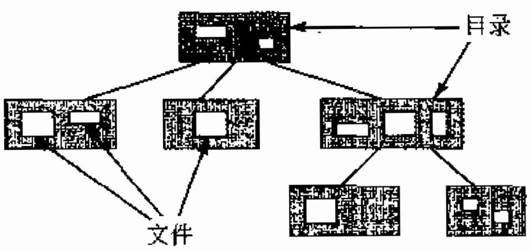


图 3.1 目录树结构

图 3.1 中的大方框表示目录，大方框内的小方框表示文件，目录之间的连线表示目录之间的组织关系。

在 Unix 系统中，每个文件都位于某个目录中，在逻辑上是没有驱动器或卷的，当然在物理上一个系统可以有多个驱动器或分区，每个驱动器上都可以有分区，位于不同驱动器和分区上的目录通过文件树无缝地连接在一起，甚至软盘、光盘这些移动存储介质也被挂到文件树的某一个子目录来处理。

这些使 ls 的实现极为简单，只需考虑文件和目录两种情况，而无需考虑驱动器或分区。

3.4 问题 2：ls 是如何工作的

ls 产生一个文件名的列表，它大致是这样工作的：

```
     open directory
+->read entry              - end of dir? -+
 __ display file info                     |
    close directory        <-----------+
```

上述逻辑与 who 的十分相似，主要的区别是 who 从文件中读数据，而 ls 从目录中读数据，读目录与读文件区别大吗？目录到底是什么呢？

3.4.1 什么是目录

先来看一看什么是目录。目录是一种特殊的文件，它的内容是文件和目录的名字。从某种程度上说，目录文件与上一章讲的 utmp 文件很类似。它们都包含很多记录，每个记录的格式由统一的标准定义。每条记录的内容代表一个文件或目录。

与普通文件不同的是，目录文件永远不会空，每个目录都至少包含两个特殊的项——“.”和“..”，其中“.”表示当前目录，“..”表示上一级目录。

3.4.2 是否可以用 open、read 和 close 来操作目录

通过以下实验来看目录文件的内容：

```
$ cat /
as'.a'asa..a'bw.tagsb'c{
quota.userc'{
        quota.group'esbetce' 'sbtmp' "sbdev" sbmnt ' wcsbin '2
   sbopt2
        '8
              sbusr8
                   '9
                        sbvar9
(many lines of hard - to - read data omitted)

$ more /tmp
```

```
/tmp is a directory

$ od -c /dev
0000000 360 001  \0  \0 024  \0 001  \0   .  \0  \0  \0 360 001  \0  \0
0000020 001 200  \0  \0 002  \0  \0  \0 024  \0 002  \0   .   .  \0  \0
0000040 002  \0  \0  \0 001 200  \0  \0 361 001  \0  \0 030  \0  \a  \0
0000060   M   A   K   E   D   E   V  \0 361 001  \0  \0 001 200  \0  \0
0000100 362 001  \0  \0 030  \0 004  \0   k   l   o   g  \0  \0  \0  \0
0000120 362 001  \0  \0 001 200  \0  \0 363 001  \0  \0 030  \0 004  \0
0000140   k   c   o   n  \0  \0  \0  \0 363 001  \0  \0 001 200  \0  \0
0000160 364 001  \0  \0 030  \0  \a  \0   k   b   i   n   l   o   g  \0
0000200 364 001  \0  \0 001 200  \0  \0 365 001  \0  \0 030  \0 004  \0
0000220   k   m   e   m  \0  \0  \0  \0 365 001  \0  \0 001 200  \0  \0
0000240 366 001  \0  \0 024  \0 003  \0   m   e   m  \0 366 001  \0  \0
```

从以上的例子中可以发现 3 点，第一，cat 和 od 可以打开目录。因为 cat 和 od 使用的是标准的文件操作系统调用，所以目录可以被 open、read、close 打开；第二，more 可以区分出文件和目录，它拒绝对目录进行操作，有些版本的 cat 和 more 一样拒绝显示目录内容；第三，从以上列出的目录内容可以知道，目录内不是无格式的文本而是包含一定的数据结构。

实际上用 open、read、close 这些系统调用来操作目录并不是很好的方法，Unix 支持多种的目录类型，有 Apple HFS、ISO9660、VFAT、NFS 等，如果用 read 来读，那么需要了解这些不同类型目录各自的结构细节。

3.4.3 如何读目录的内容

什么函数可以读目录呢？在联机帮助中根据关键字 direct 来查找答案：

```
$ man -k direct
```

我所用的系统返回了 81 条主题，用 grep 滤出那些包含 read 的主题：

```
$ man -k direct | grep read
DXmHelpSystemDisplay (3X) - Displays a topic or directory of the help file in Bookreader.
opendir, readdir, readdir_r, telldir, seekdir, rewinddir, closedir(3) - Performs operations
    on directories
$
```

其中的 readdir 正是需要的，来看它的联机帮助：

```
$ man 3 readdir
opendir(3)                                                   opendir(3)

NAME

opendir, readdir, readdir_r, telldir, seekdir, rewinddir, closedir -
Performs operations on directories
```

```
LIBRARY

Standard C Library (libc.a)
SYNOPSIS

#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dir_name);
struct dirent * readdir(DIR *dir_pointer);
int readdir_r(DIR *dir_pointer, struct dirent *entry,
    struct dirent **result);
long telldir(DIR *dir_pointer);
void seekdir(DIR *dir_pointer, long location );
void rewinddir(DIR *dir_pointer);
int closedir(DIR *dir_pointer);
[more] (11%)
```

通过联机帮助可以知道，从目录读数据与从文件读数据是类似的，opendir 打开一个目录，readdir 返回目录中的当前项，closedir 关闭一个目录，seekdir、telldir、rewinddir 与 lseek 的功能类似，如图 3.2 所示。

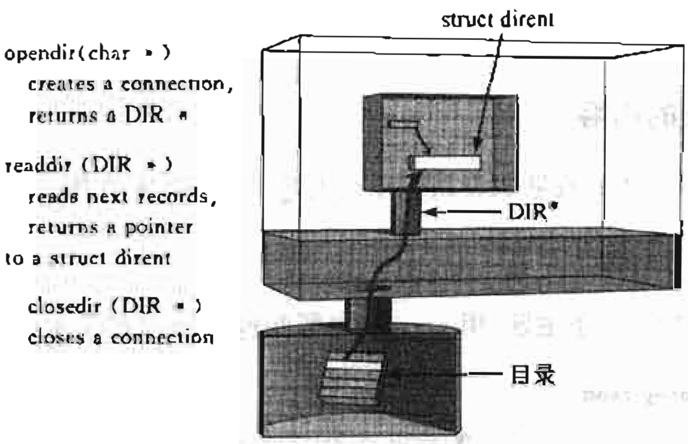


图 3.2　从目录中读到一项

目录是文件的列表，更确切地说，是记录的序列，每条记录对应一个文件或子目录。通过 readdir 来读取目录中的记录，readdir 返回一个指向目录的当前记录的指针，记录的类型是 struct dirent，这个结构定义在/usr/include/dirent.h 中，联机帮助中也可以查到。

在 SunOS 中关于它的帮助信息如下：

```
File Formats                                dirent(4)

NAME

    dirent - file system independent directory entry
```

```
SYNOPSIS
    #include <dirent.h>

DESCRIPTION
    Different file system types may have different directory entries. The dirent structure
    defines a file system independent directory entry, which contains information common to
    directory entries in different file system types. A set of these structures is returned by
    the getdents(2) system call.

    The dirent structure is defined:

        struct dirent {
            ino_t           d_ino;
            off_t           d_off;
            unsigned short  d_reclen;
            char            d_name[1]①;
    };
```

dirent 结构中成员 d_name 用于存放文件名。注意在此系统中 d_name 被定义为只有一个元素的数组，这如何能做到呢？因为一个字符的空间只能存放字符串的结束符。

3.5 问题 3：如何编写 ls

ls 的算法如下：

```
main()
   opendir
   while ( readdir )
      print d_name
   closedir
```

完整的代码如下：

```
/** ls1.c
 **   purpose list contents of directory or directories
 **   action if no args, use . else list files in args
 **/
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

void do_ls(char []);

main(int ac, char *av[])
```

① 译者注：关于 d_name 的定义，UNIX 的各个版本稍有不同，在 Linux 中是 char d_name [NAME_MAX+1]。

```
{
    if ( ac == 1 )
        do_ls( "." );
    else
        while ( --ac ){
            printf("%s:\n", *++av );
            do_ls( *av );
        }
}

void do_ls( char dirname[] )
/*
 * list files in directory called dirname
 */
{
    DIR *dir_ptr;               /* the directory */
    struct dirent *direntp;     /* each entry */

    if ( ( dir_ptr = opendir( dirname ) ) == NULL )
        fprintf(stderr,"ls1: cannot open %s\n", dirname);
    else
    {
        while ( ( direntp = readdir( dir_ptr ) ) != NULL )
            printf("%s\n", direntp->d_name );
        closedir(dir_ptr);
    }
}
```

将上述代码编译运行，并将输出与标准的 ls 的输出做比较：

```
$ cc -o ls1 ls1.c
$ ls1
.
..
s.tar
tail1
Makefile
ls1.c
ls2.c
chap03
old_src
docs
ls1
stat1.c
statdemo.c
```

```
tail1.c

$ ls
Makefile  docs  ls1.c  old_src  stat1.c     tail1
chap03    ls1   ls2.c  s.tar    statdemo.c  tail1.c
$
```

还能做什么

看来 ls 的第一个版本还不错，但是以下功能还需要加进去。

(1) 排序

ls1 的输出没有经过排序，解决办法：把所有的文件名读入一个数组，用 qsort 函数把数组排序。

(2) 分栏

标准的 ls 的输出是分栏排列的，有些以行排序输出，有些以列排序输出。解决办法：先把文件名读入数组，然后计算出列的宽度和行数。

(3) "."文件

ls1 列出了"."文件，而标准的 ls 只有在给出 -a 选项时才会列出这些文件。解决办法：使 ls1 能够接收选项 -a，并在没有 -a 的时候不显示隐藏文件。

(4) 选项 -l

如果选项里有 -l，标准的 ls 会列出文件的详细信息，而 ls1 不会。解决办法：要解决这个问题不是太容易，因为 dirent 结构中没有提供所需要的信息，如文件大小、文件所有者等。如果文件的这些信息不是存储在目录中，那么它们会存储在什么地方呢？

3.6　编　写　ls -l

ls 要做两件事情，一是列出目录的内容，二是显示文件的详细信息，这实际上是两件不同的工作，目录包含文件名，文件信息则需要从另外的途径获得，接下来从 3 个问题入手，来解决文件信息显示的问题。

3.6.1　问题 1：ls -l 能做些什么

先来看 ls -l 的输出：

```
$ ls - l
total 108
-rw-rw-r--  2  bruce  users    345  Jul  29  11:05  Makefile
-rw-rw-r--  1  bruce  Users  27521  Aug   1  12:14  chap03
drwxrwxr-x  2  bruce  Users   1024  Aug   1  12:15  docs
-rw-r--r--  1  bruce  Users    723  Feb   9   1998  ls1.c
-rw-r--r--  1  bruce  Users   3045  Feb  15  03:51  ls2.c
drwxrwxr-x  2  bruce  Users   1024  Aug   1  12:14  old_src
```

```
-rw-rw-r--   1   bruce   Users     30720   Aug    1   12:05   s.tar
-rw-r--r--   1   bruce   support     946   Feb   18   17:15   stat1.c
-rw-r--r--   1   bruce   support     191   Feb    9    1998   statdemo.c
-rwxrwxr-x   1   bruce   Users     37351   Aug    1   12:13   tail1
-rw-r--r--   1   bruce   Users      1416   Aug    1   12:05   tail1.c
-rw-r--r--   1   cse215  cscie215    574   Feb    9    1998   writable.c
$
```

每行都包含 7 个字段。

模式(mode)	每行的第一个字符表示文件类型。"-"代表普通文件,"d"代表目录,其他的类型以后还会遇到。 接下来的 9 个字符表示文件访问权限,分为读权限、写权限和执行权限,又分别针对 3 种对象:用户、同组用户和其他用户,所以一共需要 9 位来表示。从前面的 ls -l 的输出中可以看出,所有的文件和目录对所有用户都是可读的,只有文件的所有者才能对文件进行修改,所有用户都有 tail1 的执行权限。
链接数(links)	链接数指的是该文件被引用的次数,这方面的内容将在下一章介绍。
文件所有者(owner)	指出文件所有者的用户名。
组(group)	指文件所有者所在的组。有些版本的 ls 显示组名。
大小(size)	第五列显示文件的大小。在前面的 ls -l 的输出中,所有的目录大小相等,都是 1024 字节,因为目录所占空间的分配是以块(block)为单位的,每个块 512 字节,所以目录的大小经常是相等的。如果是一般的文件,size 列则显示了文件中数据的实际字节数。
最后修改时间(last-modified)	文件的最后修改时间。如果是较新的文件,会列出月、日和时刻,对于较老的文件,只能列出月、日和年。
文件名(name)	文件名

3.6.2　问题 2: ls -l 是如何工作的

如何得到文件的信息呢?在键盘上输入:

```
$ man -k file | grep -i information
```

在有些系统上可以得到有用的参考信息,有些却不可以。因为这些系统使用的术语是文件状态(file status)而不是文件信息(file information)或者文件属性(file properties)来代表文件的各种信息。提取文件状态的系统调用是 stat。

3.6.3　用 stat 得到文件信息

图 3.3 显示了 stat 的工作方式。

stat(name, ptr)

将 name 所指定的文件信息读入一个结构中。

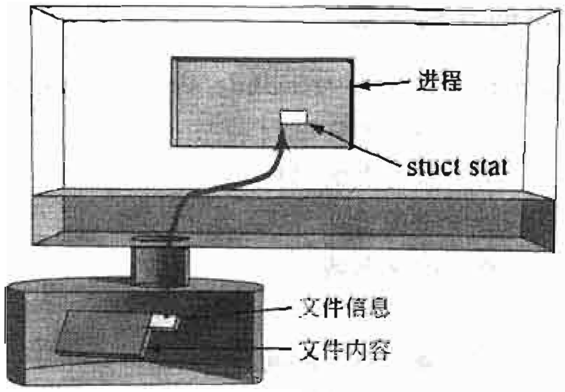


图 3.3 用 stat 读取文件的属性

磁盘上的文件有很多属性，如文件大小、文件所有者的 ID 等。如果需要得到文件属性，进程可以定义一个结构 struct stat，然后调用 stat，告诉内核把文件属性存放到这个结构中。

	Stat	
目标	得到文件的属性	
头文件	#include <sys/stat.h>	
函数原型	int result = stat(char * fname, struct stat * bufp)	
参数	fname	文件名
	bufp	指向 buffer 的指针
返回值	-1	遇到错误
	0	成功返回

stat 把文件 fname 的信息复制到指针 bufp 所指的结构中。下面的代码展示了如何用 stat 来得到文件的大小：

```
/* filesize.c - prints size of passwd file */

#include <stdio.h>
#include <sys/stat.h>

int main()
{
   struct stat infobuf;                          /* place to store info */
   if ( stat( "/etc/passwd", &infobuf) == -1 )   /* get info */
      perror("/etc/passwd");
   else
      printf(" The size of /etc/passwd is %d\n",
          infobuf.st_size );
}
```

stat 把文件的信息复制到结构 infobuf 中，程序从成员变量 st_size 中读到文件大小。

3.6.4 stat 提供的其他信息

stat 的联机帮助和头文件/usr/include/sys/stat.h 描述了 struct stat 的成员变量：

st_mode	文件类型和许可权限
st_uid	用户所有者的 ID
st_gid	所属组的 ID
st_size	所占的字节数
st_nlink	文件链接数
st_mtime	文件最后修改时间
st_atime	文件最后访问时间
st_ctime	文件属性最后改变时间

stat 结构中其他未被 ls -l 用到的成员变量未在这里列出。下面的例子 fileinfo.c 得到以上这些属性，并显示出来。

```
/* fileinfo.c - use stat() to obtain and print file properties
 *            - some members are just numbers...
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int ac, char * av[])
{
    struct stat info;               /* buffer for file info */

    if (ac>1)
        if ( stat(av[1], &info) != -1 ){
            show_stat_info( av[1], &info );
            return 0;
        }
        else
            perror(av[1]);          /* report stat() errors */
    return 1;
}

show_stat_info(char * fname, struct stat * buf)
/*
 * displays some info from stat in a name = value format
 */
{
    printf("   mode : %o\n", buf->st_mode);     /* type + mode */
    printf("  links : %d\n", buf->st_nlink);    /* # links */
    printf("   user : %d\n", buf->st_uid);      /* user id */
    printf("  group : %d\n", buf->st_gid);      /* group id */
```

```
    printf("    size : %d\n", buf->st_size);        /* file size */
    printf(" modtime : %d\n", buf->st_mtime);       /* modified */
    printf("    name : %s\n", fname );              /* filename */
}
```

编译并运行 fileinfo，并把它跟 ls -l 作对比：

```
$ cc -o fileinfo fileinfo.c
$ ./fileinfo fileinfo.c
    mode: 100664
   links: 1
    user: 500
   group: 120
    size: 1106
 modtime: 965158604
    name: fileinfo.c
$ ls -l fileinfo.c
-rw-rw-r-- 1 bruce users 1106 Aug 1 15:36 fileinfo.c
```

3.6.5 如何实现

链接数(links)、文件大小(size)的显示都没有问题，最后修改时间(modtime)是 time_t 类型的，可以用 ctime 将其转化成字符串，也没问题。

fileinfo 将模式(mode)字段以数字形式输出，然而需要的是如下的形式：

```
-rw-rw-r--
```

结构中的用户所有者(user)和组(group)字段都是数值，而显示出来的应该是用户名和组名，为了完善 ls -l，必须进一步处理模式、用户名和组的显示。

3.6.6 将模式字段转换成字符

文件类型和许可权限是如何存储在 st_mode 中？又如何将它们转成 10 个字符的串？八进制的 100664 又与“-rw-rw-r--”有什么关系呢？对这 3 个问题的回答就构成了本节的内容。

st_mode 是一个 16 位的二进制数，文件类型和权限被编码在这个数中，如图 3.4 所示。

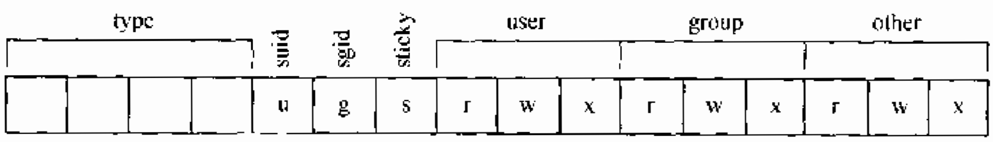


图 3.4　文件类型与许可权限

其中前 4 位用作文件类型，最多可以标识 16 种类型，目前已经使用了其中的 7 个。

接下来的 3 位是文件的特殊属性，1 代表具有某个属性，0 代表没有，这 3 位分别是 set-user-ID 位、set-group-ID 位和 sticky 位，它们的含义以后介绍。

最后的 9 位是许可权限，分为 3 组，对应 3 种用户，它们是文件所有者、同组用户和其他用户。其他用户指与用户不在同一个组的人。每组 3 位，分别是读、写和执行的权限。相应的地方如果是 1，就说明该用户拥有对应的权限，0 代表没有。

1. 字段的编码

把多种信息编码到一个整数的不同字段中是一种常用的技术，如：

	编码的例子
617 - 495 - 4204	电话号码(区号 - 局号 - 线号)
027 - 93 - 1111	社会保障号
128.103.33.100	IP 地址

2. 如何读取被编码的值

怎么来读取被编码的值呢？比如怎么知道 212 - 222 - 4444 所对应的区号是 212？很简单，一种方法是将号码的前 3 位同 212 比较，另一种方法是将暂时不需要的地方置 0，这里把电话号码的后 7 位置 0，然后同 212 - 000 - 0000 比较。

为了比较，把不需要的地方置 0，这种技术称为掩码(masking)，就如同带上面具把其他部位都遮起来，就只留下眼睛在外面。这里用一系列掩码来把 st_mode 的值转化成 ls -l 要显示的字符串。

子域编码(subfield coding)是系统编程中一种重要且常用的技术，以下从四方面详细介绍子域编码与掩码。

(1) 掩码的概念

掩码会将不需要的字段置 0，需要的字段的值不发生改变。

(2) 整数是 bit 组成的序列

整数在计算机中是以 bit 序列的形式存在的，图 3.5 显示了如何以二进制的 0 和 1 的串来表示十进制的 215。想一下 00011010 表示十进制的几？

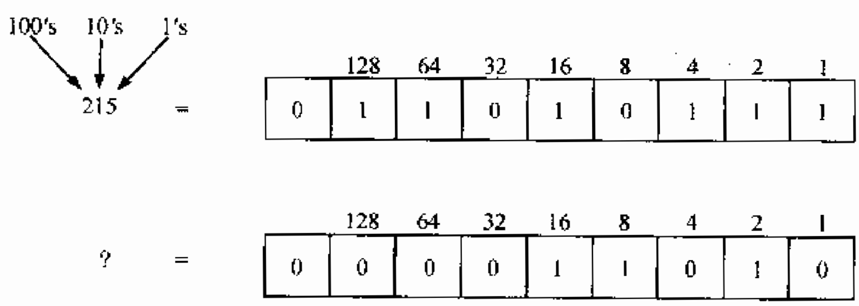


图 3.5　在整数和二进制数之间转换

(3) 掩码技术

与 0 作位与(&)操作可以将相应的 bit 置为 0，图 3.6 是八进制的 100664 通过位与操作

把一些 bit 置为 0。注意，数字中的某些 1 是如何被置为 0 的。

	1	0	0	0	0	0	0	1	1	0	1	1	0	1	0	0
&	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0
=	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0

图 3.6　位与操作

（4）使用八进制数

直接处理二进制数是很枯燥乏味的。如同处理一长串十进制数时人们常将它们三位一组分开（如 23,234,456,022）一样，一种简化的方法是将二进制数每三位分为一组来操作，这就是八进制数（0 至 7）。

如可以把二进制的 1000000110110100 分为 1,000,000,110,110,100，从而得到八进制的 100664，这样更容易理解。

3. 使用掩码来解码得到文件类型

文件类型在模式字段的第一个字节的前四位，可以通过掩码来将其他的部分置 0，从而得到类型的值。

在<sys/stat.h>中有以下定义：

```
#define    S_IFMT    0170000    /* type of file */
#define    S_IFREG   0100000    /* regular */
#define    S_IFDIR   0040000    /* directory */
#define    S_IFBLK   0060000    /* block special */
#define    S_IFCHR   0020000    /* character special */
#define    S_IFIFO   0010000    /* fifo */
#define    S_IFLNK   0120000    /* symbolic link */
#define    S_IFSOCK  0140000    /* socket */
```

S_IFMT 是一个掩码，它的值是 0170000，可以用来过滤出前四位表示的文件类型。S_IFREG 代表普通文件，值是 0100000，S_IFDIR 代表目录文件，值是 0040000。下面的代码：

```
if ((info.st_mode & 0170000) == 0040000 )
   printf("this is a directory");
```

通过掩码把其他无关的部分置 0，再与表示目录的代码比较，从而判断这是否是一个目录。

更简单的方法是用<sys/stat.h>中的宏来代替上述代码：

```
/*
 *   File type macros
 */
#define S_ISFIFO(m)  (((m)&(0170000)) == (0010000))
```

```
#define S_ISDIR(m)   (((m)&(0170000)) == (0040000))
#define S_ISCHR(m)   (((m)&(0170000)) == (0020000))
#define S_ISBLK(m)   (((m)&(0170000)) == (0060000))
#define S_ISREG(m)   (((m)&(0170000)) == (0100000))
```

使用宏的话就可以这样写代码：

```
if (S_ISDIR(info.st_mode))
   printf("this is a directory");
```

4．解码得到许可权限

模式字段的最低 9 位是许可权限，它标识了文件所有者、组用户、其他用户的读、写、执行权限。ls 将这些位转换为短横和字母的串。在<sys/stat.h>中每一位都有相应的掩码，下面的代码给出了如何使用的例子：

```
/*
 * This function takes a mode value and a char array
 * and puts into the char array the file type and the
 * nine letters that correspond to the bits in mode.
 * NOTE: It does not code setuid, setgid, and sticky
 * codes
 */
void mode_to_letters( int mode, char str[] )
{
   strcpy( str, "----------" );            /* default = no perms */

   if ( S_ISDIR(mode) ) str[0] = 'd';      /* directory? */
   if ( S_ISCHR(mode) ) str[0] = 'c';      /* char devices */
   if ( S_ISBLK(mode) ) str[0] = 'b';      /* block device */

   if ( mode & S_IRUSR ) str[1] = 'r';     /* 3 bits for user */
   if ( mode & S_IWUSR ) str[2] = 'w';
   if ( mode & S_IXUSR ) str[3] = 'x';

   if ( mode & S_IRGRP ) str[4] = 'r';     /* 3 bits for group */
   if ( mode & S_IWGRP ) str[5] = 'w';
   if ( mode & S_IXGRP ) str[6] = 'x';

   if ( mode & S_IROTH ) str[7] = 'r';     /* 3 bits for other */
   if ( mode & S_IWOTH ) str[8] = 'w';
   if ( mode & S_IXOTH ) str[9] = 'x';
}
```

5．解码并编写 ls

到此为止，已经可以正确处理文件大小、链接数、文件名、模式、最后修改时间。最后还

有一个要解决的问题是文件所有者(user)和组(group)的表示。

3.6.7 将用户/组 ID 转换成字符串

在 struct stat 中，文件所有者和组是以 ID 的形式存在的，然而 ls 要求输出用户名和组名，如何根据 ID 找到用户名和组名呢?

可以试着在联机帮助中查找关键字 username、uid、group，看看有什么结果。不同的系统中得到的结果很不相同。下面是一些说明：

(1) /etc/passwd 包含用户列表

回想一下登录过程，输入用户名和密码，经过验证后登录成功，出现提示符。系统怎么知道用户名和密码是正确的?

这就涉及到/etc/passwd 这个文件，它包含了系统中所有的用户信息，下面是一个例子：

```
root:WPA4d1OwUxypE:0:0:root:/bin/bash
bin:*:1:1:/bin:/bin:
daemon:*:2:2:daemon:/sbin:
smith:x1mEPcp4Tnokc:9768:3073:Jame Q Smith:/home/s/smith/:
      /shells/tcsh
fred:mSuVNOF4CRTmE:20359:550:Fred:/home/f/fred:/shells/tcsh
diane:7oUS8f1PsrccY:20555:550:Diane Abramov:/home/d/diane:
      /shells/tcsh
ajr:WitmEBWylar1w:3607:3034:Ann Reuter:/home/a/ajr:/shells/bash
```

这是个纯文本文件，每一行代表一个用户，用冒号":"分成不同的字段，第一个字段是用户名，第二个字段是密码，第三个字段是用户 ID，第四个字段是所属的组，接下来的是用户的全名、主目录、用户使用 shell 程序的路径。所有的用户对这个文件都有读权限，关于这个文件的详细信息，参见联机帮助。

似乎使用这个文件就可以解决用户 ID 和用户名的关联问题，只需搜索用户 ID，然后就可以得到相应的用户名。然而在实际应用中并不是这样做的，搜索文件是一件很繁琐的工作，而且对于很多网络计算系统，这种方法是不起作用的。

(2) /etc/passwd 并没有包括所有的用户

每个 Unix 系统都有/etc/passwd 这个文件，但它并没有包括所有的用户，在有些网络计算系统中，用户要能够登录到系统中的任何一台主机上，如果通过/etc/passwd 来实现，就必须在所有的主机上维护用户信息，要修改密码的话也必须修改所有主机上的密码，如果有一台主机刚好宕机了，那么在宕机期间的用户变化情况就无法同步到这台主机上。

较好的解决方法是在一台大家都能够访问到的主机上保存所有用户信息，它被称做 NIS，所有的主机通过 NIS 来进行用户身份验证。所有需要用户信息的程序也从 NIS 上获取。而本地只保存所有用户的一个子集以备离线操作。在联机帮助中有 NIS 的详细信息。

(3) 通过 getpwuid 来得到完整的用户列表

可以通过库函数 getpwuid 来访问用户信息，如果用户信息保存在/etc/passwd 中，那么

getpwuid 会查找/etc/passwd 的内容，如果用户信息在 NIS 中，getpwuid 会从 NIS 中获取信息，所以用 getpwuid 使程序有很好的可移植性。

getpwuid 需要 UID(user ID)作为参数，返回一个指向 struct passwd 的指针，这个结构定义在/usr/include/pwd.h 中：

```
/* The passwd structure */
struct passwd {
        char * pw_name;            /* Username */
        char * pw_passwd;          /* Password */
        __uid_t pw_uid;            /* User ID */
        __gid_t pw_gid;            /* Group ID */
        char * pw_gecos;           /* Real name */
        char * pw_dir;             /* Home directory */
        char * pw_shell;           /* Shell Program */
};
```

这正是 ls -l 的输出中需要的：

```
/*
 * returns a username associated with the specified uid
 * NOTE: does not work if there is no username
 */
char * uid_to_name(uid_t uid)
{
    return getpwuid(uid) ->pw_name;
}
```

这段代码很简单，但还不够健壮，如果 uid 不是一个合法的用户 ID，那 getpwuid 返回空指针 NULL，这时 getpwuid(uid)->pw_name 就没有意义，这种情况会发生吗？常用的 ls 命令有一种处理这种情况的办法。

(4) UID 没有对应的用户名

假设在一台 Unix 主机上有一个账号，用户名是 pat，用户 ID 是 2000，创建了一个文件，这个文件的 st_uid 的值就是 2000。

假设一段时间以后你搬走了，系统管理员于是把这个账号删除，在 passwd 中不再有 pat 这一行，这时如果 getpwuid 得到的参数是 2000，它就会返回 NULL。

标准的 ls 如果遇到这种情况，会打印出 UID。

当新加入一个用户时，新用户有可能与一个已被删除的用户有相同的 UID，这时，老用户所留下来的文件会被用户所拥有，新用户对这些文件具有所有的权限。

最后一个问题是组 ID 如何处理？什么是组？什么是组 ID？

(5) /etc/group 是组的列表

对一台公司里的主机而言，可能要将用户分为不同的组，如销售人员一组、行政人员一组等。要是在学校里，可能有教师组和学生组。Unix 提供了进行组管理的手段，文件

/etc/group是一个保存所有的组信息的文本文件：

```
root::0:root
other::1:
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
adm::4:root,adm,daemon
uucp::5:root,uucp
mail::6:root
tty::7:root,tty,adm
lp::8:root,lp,adm
```

第一个字段是组名，第二个是组密码，这个字段极少用到，第三个是组 ID(GID)，第四个是组中的成员列表。

(6) 用户可以同时属于多个组

passwd 文件中有每个用户所属的组，实际上那里列出的是用户的主组(primary group)。用户还可以是其他组的成员，要将用户添加到组中，只要把它的用户名添加到/etc/group 中这个组所在行的最后一个字段即可。在刚才的例子中，用户 adm 同时属于 sys、adm、tty、lp 等多个组。这个列表在处理组访问权限时会被用到。例如一个文件属于 lp 组，且组成员有这个文件的写权限，所以用户 adm 就可以修改这个文件。

(7) 通过 getgrgid 来访问组列表

在网络计算系统中，组信息也被保存在 NIS 中。Unix 系统提供 getgrgid 函数屏蔽掉实现的差异。用这个函数，用户可以得到组名而不用操心实现的细节。getgrgid 的用户手册对这个函数及相关函数做了详细解释。在 ls -l 中，可以这样得到组名：

```
/*
 * returns a groupname associated with the specified gid
 * NOTE: does not work if there is no groupname
 */
char * gid_to_name(gid_t gid)
{
    return getgrgid(gid)->gr_name;
}
```

3.6.8　编写 ls2.c

对 ls -l 的每一项输出，都有办法将它们转换为用户可理解的格式。把它们综合起来，就得到了以下代码：

```
/* ls2.c
 * purpose list contents of directory or directories
 * action if no args, use . else list files in args
 * note uses stat and pwd.h and grp.h
```

```
 * BUG: try ls2 /tmp
 */
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>

void do_ls(char[]);
void dostat(char *);
void show_file_info( char *, struct stat *);
void mode_to_letters( int , char [] );
char *uid_to_name( uid_t );
char *gid_to_name( gid_t );

main(int ac, char *av[])
{
   if ( ac == 1 )
      do_ls( "." );
   else
     while ( --ac ){
        printf("%s:\n", *++av );
        do_ls( *av );
     }
}

void do_ls( char dirname[] )
/*
 * list files in directory called dirname
 */
{
   DIR     *dir_ptr;            /* the directory */
   struct dirent *direntp;      /* each entry */

   if ( ( dir_ptr = opendir( dirname ) ) == NULL )
      fprintf(stderr,"ls1: cannot open %s\n", dirname);
   else
   {
      while ( ( direntp = readdir( dir_ptr ) ) != NULL )
         dostat( direntp->d_name );
      closedir(dir_ptr);
   }
}

void dostat( char *filename )
{
   struct stat info;
```

```
    if ( stat(filename, &info) == -1 )          /* cannot stat */
        perror( filename );                     /* say why */
    else                                        /* else show info */
        show_file_info( filename, &info );
}

void show_file_info( char * filename, struct stat * info_p )
/*
 * display the info about 'filename'. The info is stored in struct at * info_p
 */
{
    char    * uid_to_name(), * ctime(), * gid_to_name(), * filemode();
    void    mode_to_letters();
    char    modestr[11];

    mode_to_letters( info_p->st_mode, modestr );

    printf( "%s" , modestr );
    printf( "%4d " , (int) info_p->st_nlink);
    printf( "%-8s " , uid_to_name(info_p->st_uid) );
    printf( "%-8s " , gid_to_name(info_p->st_gid) );
    printf( "%8ld " , (long)info_p->st_size);
    printf( "%.12s ", 4 + ctime(&info_p->st_mtime));
    printf( "%s\n" , filename );
}

/*
 * utility functions
 */

/*
 * This function takes a mode value and a char array
 * and puts into the char array the file type and the
 * nine letters that correspond to the bits in mode.
 * NOTE: It does not code setuid, setgid, and sticky
 * codes
 */
void mode_to_letters( int mode, char str[] )
{
    strcpy( str, "----------" );          /* default = no perms */

    if ( S_ISDIR(mode) ) str[0] = 'd';    /* directory? */
    if ( S_ISCHR(mode) ) str[0] = 'c';    /* char devices */
    if ( S_ISBLK(mode) ) str[0] = 'b';    /* block device */

    if ( mode & S_IRUSR ) str[1] = 'r';   /* 3 bits for user */
```

```
    if ( mode & S_IWUSR ) str[2] = 'w';
    if ( mode & S_IXUSR ) str[3] = 'x';

    if ( mode & S_IRGRP ) str[4] = 'r';        /* 3 bits for group */
    if ( mode & S_IWGRP ) str[5] = 'w';
    if ( mode & S_IXGRP ) str[6] = 'x';

    if ( mode & S_IROTH ) str[7] = 'r';        /* 3 bits for other */
    if ( mode & S_IWOTH ) str[8] = 'w';
    if ( mode & S_IXOTH ) str[9] = 'x';
}

#include <pwd.h>

char *uid_to_name( uid_t uid )
/*
 * returns pointer to username associated with uid, uses getpw()
 */
{
    struct passwd *getpwuid(), *pw_ptr;
    static char numstr[10];

    if ( ( pw_ptr = getpwuid( uid ) ) == NULL ){
        sprintf(numstr,"%d", uid);
        return numstr;
    }
    else
        return pw_ptr->pw_name ;
}

#include <grp.h>

char *gid_to_name( gid_t gid )
/*
 * returns pointer to group number gid. used getgrgid(3)
 */
{
    struct group *getgrgid(), *grp_ptr;
    static char numstr[10];

    if ( ( grp_ptr = getgrgid(gid) ) == NULL ){
        sprintf(numstr,"%d", gid);
        return numstr;
    }
    else
        return grp_ptr->gr_name;
}
```

将 ls2 的输出与标准的 ls 对比：

```
$ ls2
drwxrwxr-x   4 bruce   bruce     1024  Aug   2  18:18  .
drwxrwxr-x   5 bruce   bruce     1024  Aug   2  18:14  ..
-rw-rw-r--   1 bruce   users    30720  Aug   1  12:05  s.tar
-rwxrwxr-x   1 bruce   users    37351  Aug   1  12:13  tail1
-rw-rw-r--   2 bruce   users      345  Jul  29  11:05  Makefile
-rw-r--r--   1 bruce   users      723  Aug   1  14:26  ls1.c
-rw-r--r--   1 bruce   users     3045  Feb  15  03:51  ls2.c
-rw-rw-r--   1 bruce   users    27521  Aug   1  12:14  chap03
drwxrwxr-x   2 bruce   users     1024  Aug   1  12:14  old_src
drwxrwxr-x   2 bruce   users     1024  Aug   1  12:15  docs
-rwxrwxr-x   1 bruce   bruce    37048  Aug   1  14:26  ls1
-rw-r--r--   1 bruce   support    946  Feb  18  17:15  stat1.c
-rwxrwxr-x   2 bruce   bruce    42295  Aug   2  18:18  ls2
-rw-r--r--   1 bruce   support    191  Feb   9  21:01  statdemo.c
-rw-r--r--   1 bruce   users     1416  Aug   1  12:05  tail1.c
$ ls -l
total 189
-rw-rw-r--   2 bruce   users      345  Jul  29  11:05  Makefile
-rw-rw-r--   1 bruce   users    27521  Aug   1  12:14  chap03
drwxrwxr-x   2 bruce   users     1024  Aug   1  12:15  docs
-rwxrwxr-x   2 bruce   bruce    42295  Aug   2  18:18  ls2
-rw-r--r--   1 bruce   users      723  Aug   1  14:26  ls1.c
-rwxrwxr-x   1 bruce   bruce    37048  Aug   1  14:26  ls1
-rw-r--r--   1 bruce   users     3045  Feb  15  03:51  ls2.c
drwxrwxr-x   2 bruce   users     1024  Aug   1  12:14  old_src
-rw-rw-r--   1 bruce   users    30720  Aug   1  12:05  s.tar
-rw-r--r--   1 bruce   support    946  Feb  18  17:15  stat1.c
-rw-r--r--   1 bruce   support    191  Feb   9   1998  statdemo.c
-rwxrwxr-x   1 bruce   users    37351  Aug   1  12:13  tail1
-rw-r--r--   1 bruce   users     1416  Aug   1  12:05  tail1.c
$
```

ls2 的输出看起来已经很不错了，模式字段、用户名和组名的处理已经完成。但还有些工作要做。标准的 ls 会显示记录总数，ls2 不会，而且 ls2 还没将结果按文件名排序，也不支持选项 -a。它还假设参数是目录名。

ls2 还有一个致命的问题，不能显示指定目录的信息，你可以试一试，在命令行输入：ls2 /tmp。可以在本章结尾的练习中修正这些问题。

3.7　三个特殊的位

结构 stat 中的 st_mode 成员包含 16 位，其中 4 位用作文件类型，9 位用作许可权限，剩下的 3 位用作文件特殊属性。

3.7.1　set - user - ID 位

在这 3 位中，第一位叫做 set - user - ID 位，它的出现是为了解决一个重要的问题，即用户如何更改自己的密码？

看起来好像很容易，用 passwd 命令就可以。

接下来研究一下 passwd 的工作原理，先来看/etc/passwd 这个文件，注意这个文件的所有者和文件访问权限设置：

```
$ ls -l /etc/passwd
-rw-r--r--  1   root   root   894   Jun   20   19:17   /etc/passwd
```

更改密码会导致上述文件内容的变化，但是普通用户没有修改这个文件的权限，只有 root 用户才可以修改它，passwd 命令怎么能够修改这个文件呢？

解决的办法，不是给所有用户修改这个文件的权限，而是给 passwd 命令一个特殊的权限，使 passwd 命令的文件所有者是 root，而且它的特殊属性中包含 set - user - ID 位，如下：

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x   1   root   bin   15725   Oct   31   1997   /usr/bin/passwd
```

SUID 位告诉内核，运行这个程序的时候认为是由文件所有者在运行这个程序，在这里就是 root，而 root 有修改/etc/passwd 的权限。

1. *是否可以更改其他用户的密码？*

答案是否定的，passwd 命令知道是谁在运行程序。它用系统调用 getuid 来得到用户 ID，passwd 命令是可以修改/etc/passwd 这个文件，但它只会修改该用户 ID 所对应的密码。

2. *set - user - ID 的其他用处*

SUID 位经常用来给某些程序提供额外的权限，比如系统中的打印队列。有可能同时有很多用户发出打印请求，但系统只有一台打印机，这时要把打印请求都放到打印队列中去，命令 lpr 负责把用户要打印的文件复制到一个特定的系统目录中去。如果这个目录所有用户都有访问权限，那将会产生一些不安全因素，恶意用户有删除别人的打印作业的可能。设置 lpr 的 SUID 位就可以解决这个问题，使 lpr 的文件所有者是 root 和 lpr。当一个普通用户调用 lpr 时，lpr 就以 root 或 lpr 的权限运行，可以对这个系统目录进行操作，而普通用户却不能直接对这个目录进行控制。执行从打印队列移除操作的程序也是设置 SUID 的。

一些游戏会把成绩最好的用户的信息写入数据库或记录文件，可以把这些执行写动作的程序的文件所有者设为数据库或记录文件的所有者，再设上 set - user - ID 位，这样普通用户都可以玩游戏，但只有游戏程序才可以修改成绩。

3. 检验 SUID 位的掩码

可以通过<sys/stat.h>中定义的掩码来检验某一个程序是否有 SUID 位：

```
#define   S_ISUID    0004000        /* set user ID on execution */
```

将它转换为二进制，将会看到它正好是 3 个特殊位的第一位。

3.7.2 set-group-ID 位

第二个特殊属性位是用来设置程序运行时所属组的，如果一个程序所属的组设为 g，而且它的 set-group-ID 位也被设置，那么程序运行的时候就好像它正被 g 组中的某一个用户运行一样。set-group-ID 位给程序某一个组的访问权限。

可以通过<sys/stat.h>中定义的掩码来检验某一个程序是否有 set-group-ID 位：

```
#define   S_ISGID    0002000        /*set group ID on execution*/
```

3.7.3 sticky 位

sticky 位对于文件和目录有不同的用途。对于文件而言，早期的 Unix 系统经常要在有限的内存中同时运行很多程序，它使用到交换(swap)技术。例如系统中内存只有 1MB 的剩余空间，但系统要同时运行 3 个程序，每个需要 0.5MB 的内存，很明显，剩余空间只允许两个程序待在内存中。在某个时候如果某个程序没运行，内核会把它临时地存放到硬盘上一个叫交换空间的分区中，空出来的内存可以给其他程序使用，当在交换空间中的程序将要再次运行时，内核会把它装载到内存中。

从交换空间装载程序要比从普通的硬盘空间快，在非交换空间的硬盘上，程序可能被分成好几块分别存放在多个地方，交换空间上的文件是不分块的。

一些经常用到的程序，如编辑器、编译器和游戏，如果位于交换空间上，那么装载的速度就会快得多。sticky 位告诉内核即使没有人在使用程序，也要把它放在交换空间中。程序粘在交换空间中，就好像口香糖粘在鞋子上一样。

现在，交换技术已经不像以前那么重要了，取而代之的是虚拟内存技术，虚拟内存使得可以以更小的单位，如页(page)，进行交换。

对于目录而言 sticky 的含义是不同的。有些目录被设计用来存放临时文件，如/tmp，谁都可以往这里创建/删除文件，sticky 位使得目录里的文件只能被创建者删除。

3.7.4 用 ls -l 看到的特殊属性

正如刚才所看到的，每个文件有 12 位的文件属性，但 ls 只用 9 个字符来表示，它是如何来表示的呢？

来看下面的例子：

```
-rwsr-sr-t   1   root   root   2345   Jun   12   14:02   sample
```

在许可权限部分，用户的 x 被替换成 s，代表 set-user-ID 被设置，组用户的 x 被替换 s，代表 set-group-ID 被设置，其他用户的 x 被替换成 t，代表 sticky 被设置，更详细的信息参见联机帮助。

3.8 ls 小 结

本章编写的 ls 程序可以列出目录里的文件，还可以列出文件的详细信息。在分析和实现的过程中，应该可以学到 Unix 很多方面的知识。

(1) 文件与目录

Unix 将数据存放于文件中，目录是特殊的文件，它的内容就是其中的文件和子目录的名字，还包含自己的名字，Unix 提供一系列函数对目录操作，打开、读、定位、关闭等，但不提供写目录的函数。

(2) 用户与组

系统中的每个用户都有用户名和用户 ID，人们可以用用户名登录到系统。系统通过 UID 来区分不同用户的文件。每个用户都属于至少一个组。每个组都有组名和组 ID。

(3) 文件属性

每个文件都有很多属性，程序通过系统调用 stat 来得到文件的属性。

(4) 文件的所有关系

每个文件都有文件所有者，文件所有者的 ID 是文件属性的一部分。同样的每个文件都属于某个组，组 ID 也是文件属性的一部分。

(5) 许可权限

文件的许可权限规定了哪种用户可以进行哪些操作，有 3 种用户：文件所有者、同组用户和其他用户，3 种操作：读、写和执行。

3.9 设置和修改文件的属性

文件有很多属性，ls -l 可以列出来，这些属性是如何建立的？是否可以改变文件的属性？如果可以，如何改？如果不可以，为什么？这是接下来要介绍的问题。

```
-rw-r--r--  1  bruce  users  3045  Feb  15  03:51  ls2.c
```

从左到右来看文件 ls2.c 的属性。

3.9.1 文件类型

文件的类型有普通文件、目录文件、设备文件、socket 文件、符号链接文件、命名管道(named pipe)文件等。

(1) 文件类型的建立

文件类型是在创建文件的时候建立的，如用系统调用 creat 建立一个普通文件。其他类型的文件如目录、设备等，可使用不同的函数创建。

(2) 修改文件类型

文件一经创建，类型就无法修改，虽然在童话里，南瓜可以变成马车，但这里的文件类型是不能变的。

3.9.2 许可位与特殊属性位

每个文件都有 9 位的许可权限和 3 位的特殊属性，它们是在文件创建的时候建立的，创建以后，它们可以被 chmod 系统调用修改。

(1) 建立文件的模式

creat 的第二个参数指定了要创建文件的许可位，如：

```
fd = creat( "newfile", 0744 );
```

指定新创建文件的许可位为 rwxr--r--。

这个参数只是请求，而不是命令。内核会通过“新建文件掩码”(file-creation-mask)来得到文件的最终模式。“新建文件掩码”是一个很有用的系统变量，它指定哪些位需要被关掉。例如要防止程序创建能被同组用户和其他用户修改的文件，那么可以通过关掉----w--w-来实现。这可以通过把“新建文件掩码”的值设为八进制数 022 来实现。例如：

```
umask( 022 );
```

这里的 umask 是一个系统命令，可以改变变量 umask 的的值。

(2) 改变文件的模式

程序可以通过系统调用 chmod 来改变文件的模式，如：

```
chmod( "/tmp/myfile", 04764) ;
chmod( "/tmp/myfile",S_ISUID|S_IRWXU| S_IRGRP|S_IWGRP|S_IROTH );
```

上述两条指令的作用相同，第一条是八进制来表示，第二条是用<sys/stat.h>中定义的符号来表示。后者有明显的优点，当系统定义的许可位的值改变时，无需修改程序。系统调用 chmod 不受“新建文件掩码”的影响。

	chmod
目标	**修改文件的许可权限和特殊属性**
头文件	#include <sys/types.h> #include <sys/stat.h>
函数原型	int result = chmod(char * path, mode_t mode);
参数	path　文件名 mode　新的许可权限和特殊属性
返回值	−1　遇到错误 0　成功返回

(3) 用来修改文件的许可权限和特殊属性的命令

Shell 命令 chmod 也可以用来完成上述操作。它可以通过两种模式指定权限和属性，八进制模式(如 04764)和符号模式(如 u=rws g=rw o=r)。

3.9.3 文件的链接数

关于链接数的详细讨论在下一章。简而言之，链接数就是文件被引用的次数(别名的数量)。如果一个文件在目录树中一共有 3 个别名，那么这个文件的链接数就是 3。增加文件的别名(使用 link)会使链接数增加，减少别名(使用 unlink)会使链接数减少。

3.9.4 文件所有者与组

每个文件都有文件所有者，Unix 通过用户 ID 和组 ID 来标识文件所有者和文件所属的组。

(1) 文件所有者

简单的说，文件所有者就是创建文件的用户，用户通过 creat 建立文件时，内核把文件所有者设为运行程序的用户，如果程序具有 set-user-ID 位，那么新文件的文件所有者就是程序的文件所有者。

(2) 组

通常情况下，新文件的组被设为执行创建动作的用户所在的组，在有些情况下，组会被设为与父目录的组相同。这听起来好像婴儿的国籍由出生地决定而不由父母的国籍来决定。

(3) 修改文件所有者和组

通过系统调用 chown 来修改文件所有者和组：

```
chown( "file1", 200, 40);
```

将文件 file1 的用户 ID 改为 200，组 ID 改为 40，如果后两个参数的值都是-1，那么文件所有者和组都不会改变。一般用户不大会修改文件的文件所有者和组，但 root 经常出于管理上的目的要修改这些内容。文件所有者可以把文件的组改成任何一个他所属的组。

	chown
目标	修改文件所有者和组
头文件	#include <unistd.h>
函数原型	int chown(char * path, uid_t owner, gid_t group)
参数	path 文件名 owner 新的文件所有者 ID group 新的组 ID
返回值	-1 遇到错误 0 成功返回

(4) 用来修改文件所有者和组的命令

Shell 命令 chown 和 chgrp 可以用来修改文件所有者和组，它们可以一次修改多个文件，

可以用用户名/组名作为参数，也可以用用户 ID/组 ID 作为参数，关于它们的详细使用说明参见联机帮助。

3.9.5 文件大小

文件、目录和命名管道的大小是它们实际所占用的存储空间的字节数目。当向文件添加内容时，文件的大小会自动增加，可以使用系统调用 creat 把文件的大小置为 0。不存在能够直接减小文件占用的空间的函数。

3.9.6 时间

每个文件都有 3 个时间：最后修改（modification）时间、最后访问（access）时间和属性（如用户所有者 ID、许可权限）最后修改时间，当文件被操作时，内核会自动地修改这些时间，也可以编程来修改最后修改时间和最后访问时间。

（1）修改最后修改时间和最后访问时间

utime 系统调用可以用来设置最后修改时间和最后访问时间，使用一个包含两个 time_t 结构的变量，一个 time_t 用来存放更新的最后修改时间，另一个是最后访问时间。

	utime	
目标	修改文件最后修改时间和最后访问时间	
头文件	#include <sys/time.h> #include <utime.h> #include <sys/types.h>	
函数原型	int utime(char * path, struct utimbuf * newtimes)	
参数	path	文件名
	newtimes	指向结构变量 utimbuf 的指针 详见 utime.h
返回值	−1	遇到错误
	0	成功返回

什么时候需要修改这些时间呢？举个例子，在文件备份的时候，文件的最后修改时间和访问时间会被记录下来，当文件被恢复的时候，希望文件的这些时间与原来的相同，这时候就可以用 utime。恢复时做了两件事，一是把备份的文件复制回去，二是把最后修改时间和访问时间改成备份时候的情况，这样被恢复的文件就与备份时的完全一样。

（2）用命令修改最后修改时间和最后访问时间

shell 命令 touch 可以修改文件的最后访问时间和最后修改时间，详细的信息参见联机帮助。

3.9.7 文件名

创建文件时会指定一个文件名，命令 mv 可以改变一个文件的名字，也可以把文件从一个地方移动到另一个地方。

（1）文件名的建立

系统调用 creat 在指定文件模式的同时会指定文件的名字。

(2) 修改文件名

系统调用 rename 可以修改文件/目录的名字,还可以移动文件的位置,它有两个参数,原文件名和新文件名。

	rename	
目标	修改文件名或移动文件的位置	
头文件	#include <stdio.h>	
函数原型	int result = rename(char *old, char *new)	
参数	old	原来的文件名或目录名
	new	新的文件名或目录名
返回值	−1	遇到错误
	0	成功返回

小　结

1. 主要内容

- 磁盘上有文件和目录,文件和目录都有内容和属性。文件的内容可以是任意的数据,目录的内容只能是文件名/子目录名的列表。
- 目录中的文件名/子目录名指向文件和其他的目录,内核提供了系统调用来读取目录的内容、读取和修改文件的属性。
- 文件类型、文件的访问权限和特殊属性被编码存储在一个 16 位整数中,可以通过掩码技术来读取这些信息。
- 文件所有者和组信息是以 ID 的形式保存的,它们与用户名和组名的联系保存在 passwd 和 group 数据库中。

2. 进一步的问题

从本章可以知道目录的内容是文件名和目录名的列表,目录被互相连接组成一棵树,它们是如何连在一起的? 下一章会对目录的结构做详细的介绍。

3. 图示

磁盘、目录、文件及它们的属性如图 3.7 所示。

4. 习题

3.1　在 struct dirent 中,数组 d_name[]的长度在有的系统上是 1,而在有的系统上是 255,实际的长度是多少? 为什么会有这些不同? 为什么不定义成 char *?

3.2　文件保护模式-------rwx 可以通过命令 chmod 007 filename 得到,虽然这种做法很奇怪,但却是合法的,其他用户对文件有全部权限,组用户和文件所有者都没有给任何权限。对于这样的文件,谁可以读? 当内核执行 open 时,采用的是什么逻辑来判断是否可以执行? 通过实验来验证自己的想法,如果可以看到内核的代码,阅读相关代码。

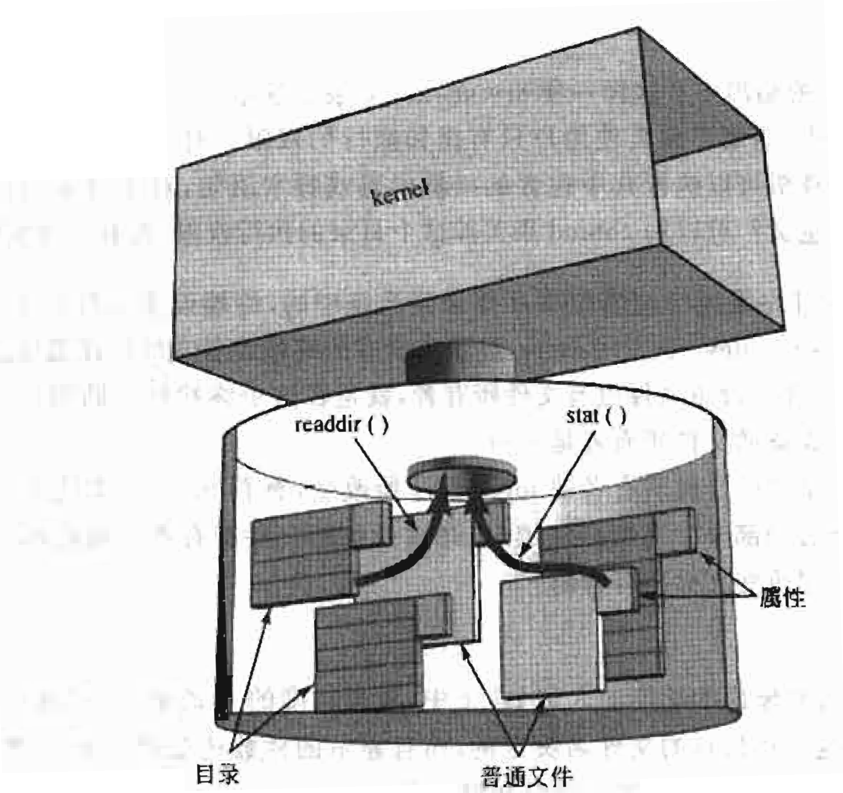


图 3.7　磁盘上有目录、文件及它们的属性

3.3　每个用户都有用户名，每个用户名都有对应的用户 ID，是否可能两个不同的用户名对应一个相同的 ID？是否允许同一个用户拥有两个不同的 ID？如果有 root 权限，可以试一试，创建两个用户，改成同一个 ID，但有不同的用户名和密码。这两个用户是否可以修改对方的文件？who 输出什么？ls -l 输出什么？命令 id 输出什么？相互发送 e-mail 呢？从中能够看出些什么？多个用户使用同一个 ID 还有什么其他用途？

3.4　与普通的文件一样，目录也有特殊属性位，其中包含 set-user-ID 和 set-group-ID 位，使 set-user-ID 有效对目录有什么影响？如果有，那么是什么？为什么？如果没有影响，那么你能想象出这些位有什么作用吗？

3.5　每个文件的执行权限都可以被打开或关闭，假设一个纯文本文件具有执行权限，它是否可以被执行？如果一个包含可执行代码的文件，如对 C 语言编译后的可执行文件 a.out，没有执行权限的话，它是否可以被执行？讨论执行权限和可执行代码之间的区别。它们之间有关系吗？参见命令 file 的联机帮助。

3.6　每个用户都有用户名和用户 ID，这可以被认为是两种标识系统，为什么要这样？能不能直接用用户名来表示文件所有者？为什么？能不能只用其中一种标识系统？两套表示系统有什么优缺点？如果由你来设计系统，你会如何设计？

3.7　命令 dirent 的联机帮助中提到了系统调用 getdents(2)，它的功能是什么？与

readdir 有什么关系？

3.8 命令 ls -l 的输出中有这样一项 drwxr-xr-x，表示这是一个目录，它的文件所有者有全部权限，组用户和其他用户只有读和执行的权限。对于文件而言，执行权限意味着计算机可以执行其中包含的机器代码或脚本语句，对于目录而言，执行权限有什么意义？可以用 chmod 来关掉这个目录的执行权限，看看会发生什么。

3.9 用户是通过终端或终端模拟程序登录到系统中的，终端设备文件位于/dev 目录下，输入 ls -l /dev/tty* | more，显示出所有的终端设备文件的详细信息，与普通文件一样，终端设备文件也有文件所有者，就是在这个终端登录的用户，无用户使用的终端设备的文件所有者是 root。
终端设备文件的文件所有者被 login 程序所改变，查看 login 的源代码，找到改变文件所有者的部分。当用户注销时，终端设备的文件所有者会被改回成 root，是哪一个程序改变文件所有者的？

5. 编程练习

3.10 为了将分栏输出的功能加入到 ls1.c 中，观察标准的 ls 的输出，发现每一栏的宽度是由这一栏最长的文件名决定的，而且显示的栏数还受终端显示器的宽度影响，每一列尽可能的等宽。这是如何实现的？

3.11 前面提到 ls2.c 无法处理在命令行给出目录作为参数(ls2 /tmp)，修改 ls2.c 使之能够处理。

3.12 修改 ls2.c，使之能够正确显示文件特殊属性 suid、sgid 和 sticky，参见联机帮助确保程序能处理各种情况。

3.13 使用标准的 cp 时，如果第二个参数是目录，那么会把第一个参数指定的文件复制到相应的目录下，如：

```
$ cp file1 /tmp
```

等价于：

```
$ cp file1 /tmp/file1
```

修改第 2 章的 cp1.c 使之能够完成上述工作。

3.14 有时需要对整个目录作备份，修改 cp1.c 使得当两个参数都是目录时，把第一个目录中的所有文件复制到第二个目录中，文件名不变。

3.15 修改 ls1.c 使得它能够将文件排序后输出。标准的 ls 支持选项 -r，它的作用是逆序输出，把这项功能也加入到 ls1.c 中。有些版本的 ls 支持选项 -q，它的作用是不排序输出，当目录中的文件特别多的时候，可以加快 ls 的输出速度。

3.16 有一个目录，它的许可权限为 rwxr-x--x，写出对应的 st_mode 的二进制形式。

3.17　当关掉一个文件的读权限，就不能打开这个文件来读。如果从一个终端登录，打开一个文件，保持文件的打开状态，然后从另外的终端登录，去掉文件的读权限，这时有什么事情会发生？编写一个程序，先用 open()打开一个文件，用 read()读一些内容，调用 sleep(20)等待 20 s 以后，再读一些内容，从另外的终端，在等待的 20 s 内去掉文件的读权限，这样会有什么结果？

3.18　标准的 ls 支持选项 -R，它的功能是递归地列出目录中所有的文件包含子目录中的文件。修改 ls2.c，使之支持这一功能。

3.19　标准的 ls 支持选项 -u，它会显示出文件的最后访问时间，如果用了 -u 而不用 -l，会有什么结果？修改 ls2.c 使之支持这一功能。提示：读 -t 选项的内容。

3.20　自己编写一个 chown，使之能够接收用户名或用户 ID 作为参数，能够一次修改多个文件的文件所有者。考虑以下问题，如何将文件名转换为用户 ID？如果用户名不存在怎么办？提示：要测试这个程序，可能需要管理员的权限。

3.21　在做文件恢复的时候，不仅希望将文件恢复，还希望将文件的最后修改时间和最后访问时间恢复，编写 cp 来实现上述要求。

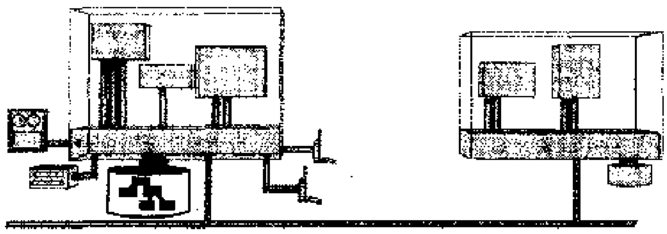
3.22　可以通过终端设备文件来与用户收发数据，如程序从这个文件读数据，就等于从用户的键盘读数据，写数据就等于将数据显示在屏幕上。struct stat 中的成员变量 st_mtime 记录了文件最后修改时间，编写程序 lastdata 列出每个用户的中断设备文件的最后修改时间，输出格式参照 who。

6. 项目

根据本章所学的内容，可以编写以下的 Unix 程序：

chmod、file、chown、chgrp、finger、touch

第 4 章　文件系统：编写 pwd



概念与技巧

- Unix 树状文件系统的概念
- Unix 文件系统的内部结构：i-节点和数据块
- 目录的连接方式
- 硬链接和符号链接的概念及相应的系统调用
- pwd 的工作原理
- 文件系统的装载(mounting)

相关系统调用

- mkdir、rmdir、chdir
- link、unlink、rename、symlink

相关命令

- pwd

4.1 介　绍

文件包含数据，而目录是文件的列表。不同的目录互相连接构成树状的结构。目录还可以包含其他的目录。文件"在一个目录中"是什么意思呢？当登录到一台 Unix 的机器上，可以说你处在"你的主目录中"，一个人"处在某个目录中"又是什么意思呢？

树状结构像是一个神话。一个硬盘实际上是由一些金属圆盘构成的，每个盘面上都有磁性物质。这些金属盘如何显示为一个包含文件、属性和目录的树状结构呢？

为回答这些问题需要编写自己的 pwd 命令。pwd 显示你在目录树中的当前位置。从树根到你所处位置所经过的目录的序列被称做路径(path)。要编写 pwd，必须了解文件和目录是如何组织和存储的。本章将先察看文件系统的外在特征，接下来分析它的内部结构，最后学习相应的系统调用的功能和使用方法。

4.2　从用户的角度看文件系统

4.2.1　目录和文件

从用户的角度来看，Unix 系统中硬盘上的文件组成一棵目录树。每个目录能包含文件或其他的目录。图 4.1 是树的一个示例。

下面将从构建这个目录结构开始，介绍管理这些文件和目录树的 Unix 命令。

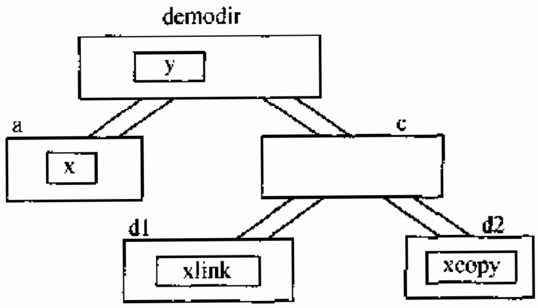


图 4.1　目录树

4.2.2　目录命令

可以按照下面的命令顺序建立如图 4.1 所示的这棵树：

```
$ mkdir demodir
$ cd demodir
$ pwd
/home/yourname/experiments/demodir
$ mkdir b cops
$ mv b c
$ rmdir cops
$ cd c
$ mkdir d1 d2
$ cd ../..
$ mkdir demodir/a
```

其实要达到同样的效果，本可以使用若干其他的命令，这里为了演示而故意做得复杂些。

按照上面的例子建立这棵树。例子中使用了一些基本的命令，mkdir 用来创建一个指定的目录或多个目录。思考以下问题，创建一个和已有文件或目录同名的目录将会怎样？rmdir 用来删除一个目录或多个目录，删除一个包含子目录的目录会发生什么事呢？mv 用来重命名一个目录，也可用来将一个目录从一个地方移到另一个地方。

与上述命令不同的是 cd 命令不对目录产生影响，它影响的是用户。cd 使你从一个目录转到另一个目录，就如同你从一个房间走到另一个房间。pwd 打印出当前工作目录。在上面的例子中，demodir 是 experiments 的一个子目录，而 experiments 位于 yourname 之下，yourname 位于 home 之下，home 位于根目录之下，根目录用“/”表示。

4.2.3　文件操作命令

现在在这棵目录树中创建一些文件：

```
$ cd demodir
$ cp /etc/group x
$ cat x
```

```
root::0:
bin::1:bin,daemon
users::200:
$ cp x copy.of.x
$ mv copy.of.x y
$ cd c
$ cp ../a/x d2/xcopy
$ ln ../a/x d1/xlink
$ ls > d1/xlink
$ cp d1/xlink z
$ rm ../../demodir/c/d2/../z
$ cd ../..
$ cat demodir /a/x
(想想接着会显示出什么?)
```

为了演示各种文件操作命令,特意设计了上述命令操作序列,请按照上面的步骤自己建立文件,想一下最后一个命令会产生怎样的结果。这个例子中用了一些最常用的文件处理命令。cp 用来复制一个文件,在前面的章节中已经编写了一个 cp 的简易版本。cat 命令将文件内容复制到标准输出文件。mv 命令可以重命名一个文件,就像在第一个例子中那样;也可将文件移动到另一个目录,就像在第二个例子中那样。rm 命令删除一个文件,请注意目录路径可能包含".."及多个目录元素。符号".."表示上一级目录,即父目录。一系列用单斜杠分隔的目录名指出了能到达指定对象的一条路径,注意此时并未改变用户所处的位置,上例中采用这种间接的方法删除了文件"z"。

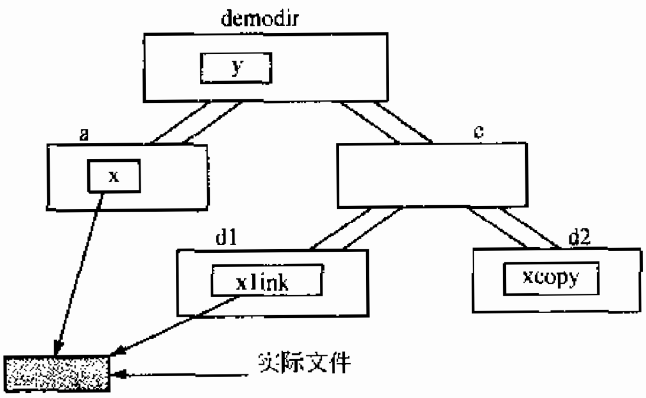


图 4.2　对同一文件的两个链接

文件的复制、显示文件的内容和重命名是任何计算机系统都会提供的操作,而命令 ln 则不是很常见,但它却是 Unix 的一个基本操作。如在上例中生成了对一个已存在文件../a/x 的一个链接,这个链接称为 d1/xlink。如图 4.2 所示,称为 x 的项位于目录 demodir/a 中,称为 xlink 的项位于目录 demodir/c/d1 中。x 和 xlink 都称为链接。一个链接是指向文件的一个指针。../a/x 和 d1/xlink 都指向硬盘上同一数据块。上例中,下一个命令 ls > dl/xlink 将 ls 的输出作为文件 xlink 的内容。那么当执行 cat ../a/x 时将会发生什么呢?

4.2.4　针对目录树的命令

有些 Unix 命令是对整个树结构进行操作，以下是一些例子：

```
ls -R
```

ls 命令用来列出目录的内容，选项 -R 要求列出指定目录及其子目录的所有内容。在前面的章节中已经给出了一个 ls 的版本，但要完成选项 -R 要求的功能，还需要做一些工作。

```
chmod -R
```

chmod 命令用来修改文件的许可权限位，选项 -R 要求修改子目录中所有文件的许可权限。

```
du
```

du 是 disk usage（硬盘使用）的缩写，该命令给出指定目录及其子目录下所有文件占用硬盘中数据块的总数。

```
find
```

find 命令将在一个目录及其所有子目录中检索符合要求的文件和目录。例如，可以检索一棵目录树中所有大于 1MB、上周末被修改过、可被所有用户阅读的文件。

Unix 中目录树是文件系统的一个重要组成部分，其他有很多命令都跟目录树有关，读者可以自己试着找一找。

4.2.5　目录树的深度几乎没有限制

目录能够包含多个文件和子目录。系统并未对目录树的深度加以限制。但是，有可能所建的目录树太深以至超过许多命令允许的范围。

注意：如果你想尝试以下的试验，请在自己的机器上做。如果在学校或工作地点进行试验，那里的系统管理员肯定会不高兴。

这是一个简单的命令脚本（关于脚本的解释详见第 8 章）。

```
while true
do
   mkdir deep-well
   cd deep-well
done
```

即使你在程序运行 1、2 秒后就按下 Ctrl-C，它也会创建一个非常深的级联目录树。那么 du 命令对这个级联目录会产生什么效果？find 和 ls -R 命令又会如何呢？

在 Unix 的很多版本中，下述命令 rm -r deep-well 将不起作用，考虑一下如何才能删除如此深的目录结构。

4.2.6 Unix 文件系统小结

这个小节中从用户的角度观察了 Unix 的文件系统。硬盘上呈现了一个能够在深度和宽度上广泛延伸的目录树结构，Unix 提供了很多命令来和这种结构的对象协同工作。Unix 系统中的所有文件都存在于这种结构中。

它们是如何工作的？目录是什么？如何知道文件所处的目录？从一个目录转换到另一个目录意味着什么？pwd 如何得知你当前所处的位置？这些问题将引导下面的学习。

4.3 Unix 文件系统的内部结构

硬盘实际上是由一些磁性盘片组成的计算机系统的一个设备。前面章节中所提及的文件系统是对该设备的一种多层次的抽象。

4.3.1 第一层抽象：从磁盘到分区

一个磁盘能够存储大量的数据。就像一个国家能被划分成州或县，一个磁盘可被划分成分区，以便在一个大的实体内创建独立的区域。每个分区都可以看作是一个独立的磁盘。

4.3.2 第二层抽象：从磁盘到块序列

一个硬盘由一些磁性盘片组成。每个盘片的表面都被划分为很多同心圆，这些同心圆称作磁道，每个磁道又进一步被划分成扇区，就像郊外的街道被划分成居住单元。每个扇区可以存储一定字节数的数据，例如每个扇区有 512 字节。扇区是磁盘上的基本存储单元，现在的磁盘包含大量的扇区。图 4.3 显示了序列号是如何分配给磁盘块的。

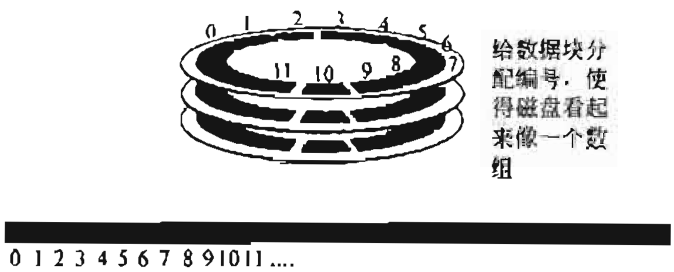


图 4.3 为数据块分配编号

为磁盘块编号是一种很重要的方法。给每个磁盘块分配连续的编号使得系统能够计算磁盘上的每个块。可以一个磁盘接一个磁盘地从上到下给所有的块编号，还可以一个磁道接一个磁道地从外向里给所有的块编号。就像给每条街道上的每所房子编号一样，磁盘上存储数据的软件给磁盘上每条磁道上的每个块分配了一个序号。

一个将磁盘扇区编号的系统使得我们可以把磁盘视为一系列块的组合。

4.3.3 第三层抽象：从块序列到三个区域的划分

文件系统可以用来存储文件内容、文件属性（文件所有者、日期等）和目录，这些不同类

型的数据是如何存储在被编号的磁盘块上的呢？

Unix 使用了一个简单的方法。如图 4.4 所示，它将这些磁盘块分成了 3 部分。

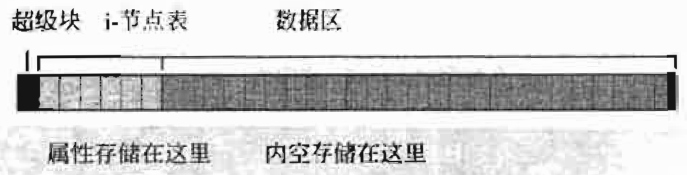


图 4.4　文件系统的三个区域

一部分称为数据区，用来存放文件内容。另一部分称为 i-节点表(inode table)，用来存放文件属性。第三部分称为超级块(superblock)，用来存放文件系统本身的信息。文件系统由这 3 部分组合而成，其中任一部分都是由很多有序磁盘块组成的。

(1) 超级块

文件系统中的第一个块被称为超级块。这个块存放文件系统本身的结构信息。例如，超级块记录了每个区域的大小。超级块也存放未被使用的磁盘块的信息。不同版本 Unix 的超级块的内容和结构稍有不同，可以察看联机帮助和头文件，以确定你的系统的超级块所包含的内容。

(2) i-节点表

文件系统的下一个部分被称为 i-节点表。每个文件都有一些属性，如大小、文件所有者和最近修改时间等。这些性质被记录在一个称为 i-节点的结构中。所有的 i-节点都有相同的大小，并且 i-节点表是这些结构的一个列表。文件系统中的每个文件在该表中都有一个 i-节点。如果你有 root 权限，就可以像操作文件一样将分区打开、阅读并显示 i-节点表。在显示 utmp 文件时就用过类似的技术。

以下这一点很重要：表中的每个 i-节点都通过位置来标识。例如，标识为 2 的 i-节点(inode 2)位于文件系统 i-节点表中的第 3 个位置。

(3) 数据区

文件系统的第 3 个部分是数据区。文件的内容保存在这个区域。磁盘上所有块的大小都是一样的。如果文件包含了超过一个块的内容，则文件内容会存放在多个磁盘块中。一个较大的文件很容易分布在上千个独立的磁盘块中。那么，系统是如何跟踪这些独立的磁盘块呢？

4.3.4　文件系统的实现：创建一个文件的过程

文件的内容和属性分区存放的想法看起来很简单，但实际上它是如何工作的呢？创建一个新文件的时候又会发生什么？考虑以下命令：

```
$ who > userlist
```

当这个命令完成时，文件系统中增加了一个存放命令 who 输出内容的新文件。这是怎么回事？

文件有内容和属性，内核将文件内容存放在数据区，文件属性存放在 i-节点，文件名存放在目录。图 4.5 显示了创建一个文件的例子，这个新文件需要 3 个存储块来存放各部分的数据。

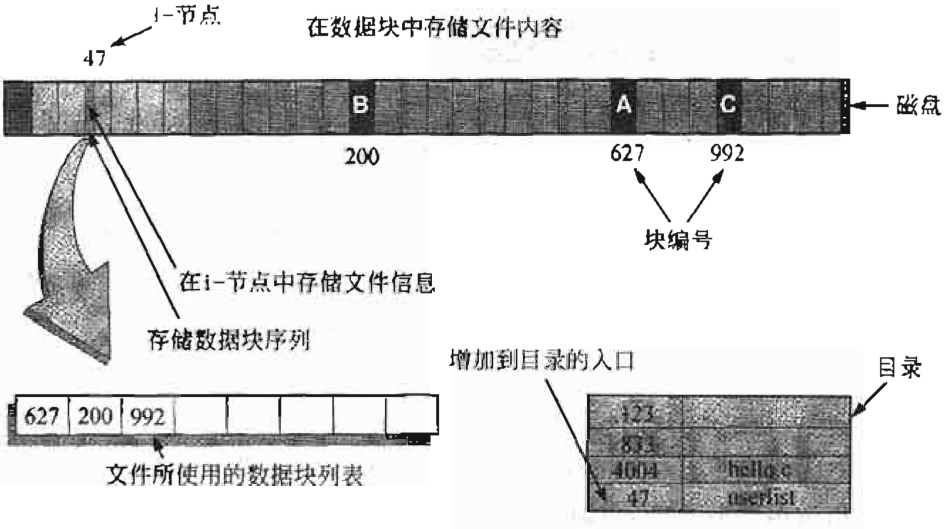


图 4.5 文件的内部结构

创建一个新文件的 4 个主要操作如下。

(1) 存储属性

文件属性的存储：内核先找到一个空的 i-节点。图 4.5 中，内核找到 i-节点 47。内核把文件的信息记录其中。

(2) 存储数据

文件内容的存储：由于该新文件需要 3 个存储磁盘块，因此内核从自由块的列表中找出 3 个自由块。图 4.5 中，它找到块 627、200 和 992。内核缓冲区的第一块数据复制到块 627，下一块数据复制到块 200，最后一块数据复制到块 992。

(3) 记录分配情况

文件内容按顺序存放在块 627、200 和 992 中。内核在 i-节点的磁盘分布区记录了上述的块序列。磁盘分布区是一个磁盘块序号的列表，这 3 个编号放在最开始的 3 个位置。

(4) 添加文件名到目录

新文件的名字是 userlist。Unix 如何在当前的目录中记录这个文件？答案很简单。内核将入口(47，userlist)添加到目录文件。文件名和 i-节点号之间的对应关系将文件名和文件的内容及属性连接了起来。这个问题将在下面进一步地讨论。

4.3.5 文件系统的实现：目录的工作过程

目录是一种包含了文件名字列表的特殊文件。不同版本的 Unix 目录的内部结构不同，但是它们的抽象模型总是一致的——一个包含 i-节点号和文件名的表。

i-节点号	文件名
2342	.
43989	..
3421	hello.c
533870	myls.c

(1) 探讨目录内部

可以通过命令 ls -1ia(选项第一位是数字 1)来看目录的内容：

```
$ ls -1ia demodir
177865 .
529193 ..
588277 a
200520 c
204491 y
$
```

输出的是文件名和对应的 i-节点号。例如，文件名 y 对应于 i-节点号 204491。当前目录用“.”表示，i-节点号是 177865。这意味着有关大小、文件所有者、组等各项关于当前目录的信息存放在 i-节点表中的编号为 177865 的结构中。

ls 的选项 -i 和 -1(数字 1 而不是字母 l)可能有点陌生。选项 -i 告诉 ls 在列表中包含 i-节点号，选项 -1 要求每行列出一个文件，在 demodir 版本上尝试这个命令，以查看所得到的 i-节点号。

(2) 指向同一文件的多重链接

可以使用命令 ls -i 查看系统上任何一个文件的 i-节点号。例如，可以查看系统上根目录中各文件的 i-节点号：

```
$ ls -ia /
      2 .          28673 etc          11 lost+found   438292 shlib
      2 ..        311297 home       4097 mnt           40961 tmp
      3 auto        8832 home2    108545 opt           18433 usr
  26625 bin        24646 initrd        1 proc          10241 var
 403457 boot       24579 install   24681 root            183 xfer.log
 225281 dev       161797 lib      233473 sbin            183 transfers
```

这个列表含有两个重要的例子。第一，在右下角有被称为 xfer.log 和 transfers 的两个文件。这两个文件都拥有 i-节点号 183。因此，两个文件都指向同一个 i-节点。i-节点实际上代表了一个文件，i-节点包含了文件的属性和数据块的列表。因此，xfer.log 和 transfers 是同一个文件的两个不同名字。这有点像电话簿里的两个不同的电话号码所对应的电话机有可能在同一所房子①。

① 电话簿的比喻不是完全恰当，因为两个不同的人都可能住在那个房子。请在网络编程章节中了解 port 的概念。

在根目录中另一个重要的例子是左上角的“.”和“..”,这两项都有 i-节点号 2,所以“.”和“..”都指向同一个目录。当前目录怎么会和父目录相同呢?[①] 实际上在大多数情况下,它们是不同的,根目录比较特别,当用 Unix 命令 mkfs 创建了一个文件系统,mkfs 将根目录的父目录指向自己。

4.3.6 文件系统的实现:cat 命令的工作原理

现在已经看到了创建一个新的文件时内部所发生的事情,例如 who > userlist。当读取一个文件时又会发生什么呢?读取命令如何工作?对如下的命令:

```
$ cat userlist
```

采用从目录文件一步一步找到数据的方法来加以了解。

(1) 在目录中寻找文件名

文件名存储在目录文件中。内核在目录文件中寻找包含字符串 userlist 的记录。userlist 所在的记录包含编号为 47 的 i-节点号,如图 4.6 所示。

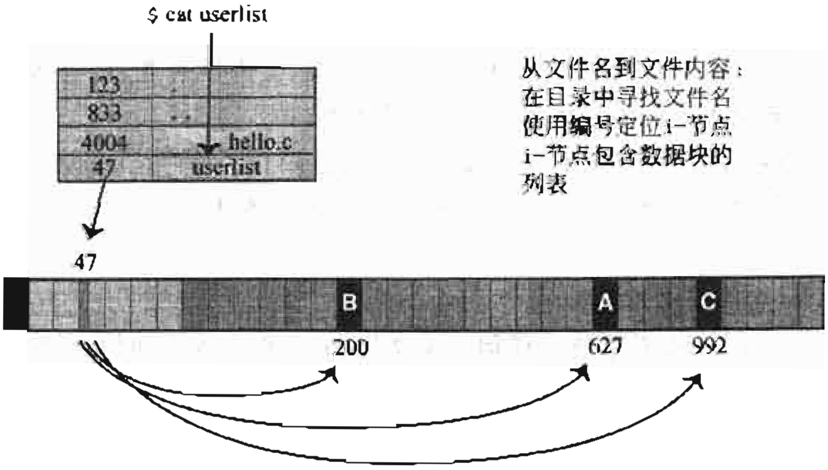


图 4.6　从文件名到磁盘块

(2) 定位 i-节点 47 并读取其内容

内核在文件系统中的 i-节点区域找到 i-节点 47。定位一个 i-节点可能需要一些简单的计算,所有的 i-节点大小相同,每个磁盘块都包含相同数量的 i-节点。为了提高访问效率,内核有可能将 i-节点置于缓冲区中。i-节点包含数据块编号的列表。

(3) 访问存储文件内容的数据块

通过以上过程,内核已经可以知道文件内容存放在哪些数据块上,以及它们的顺序。由于 cat 不断地调用 read 函数,使得内核不断将字节从磁盘复制到内核缓冲区,进而到达用户空间。

所有从文件读取数据的命令,例如 cat、cp、more、who 等,都是将文件名传给 open 来访

① 参考 Latham & Jaffe 编著的 I'm My Own Grandpa. 1947 对相关主题的讨论。

问文件内容。对 open 的每次调用都是先在目录中寻找文件名、然后根据目录中的 i-节点号获得文件的属性，最终找到文件的内容。

现在可以想象一下在 open 一个没有读或写权限的文件时将发生什么情况。内核首先根据文件名找到 i-节点号，然后根据 i-节点号找到 i-节点。在 i-节点中，内核找到文件的权限位和拥有者的用户 ID。如果权限位设置你的用户 ID 对文件没有访问权限，则 open 返回 −1 并且将全局变量 errno 的值设为 EPERM。

通过对目录、i-节点和数据块的描述，相信能提高对其他的文件操作的理解。可以通过阅读一些版本的 Unix 系统源代码来加以检验。

4.3.7　i-节点和大文件

Unix 文件系统如何跟踪大文件呢？其实前一个章节中的解释并不完整。简短地说，问题是：

事实 1	一个大的文件需要多个磁盘块
事实 2	在 i-节点中存放有磁盘块分配列表
问题	一个固定大小的 i-节点如何存储较长的分配列表？
解决方案	将分配列表的大部分存储在数据块，在 i-节点中存放指向那些块的指针。

考虑图 4.7 中描述的解决方案。这个文件需要 14 个数据块存储它的内容。因此，分配链表包含 14 个块的编号。但是很遗憾，文件的 i-节点只包含一个含有 13 个项的分配链表。14 个编号如何放到 13 个项中呢？其实很简单。将分配链表中的前 10 个编号放到 i-节点中，将最后 4 个编号放到一个数据块中。这有点像把某些货物放在架子上而把剩下的放在仓库里。

更具体地说，就是该 i-节点的链表包含分配 13 个块编号的空间，链表里的前 10 个项像

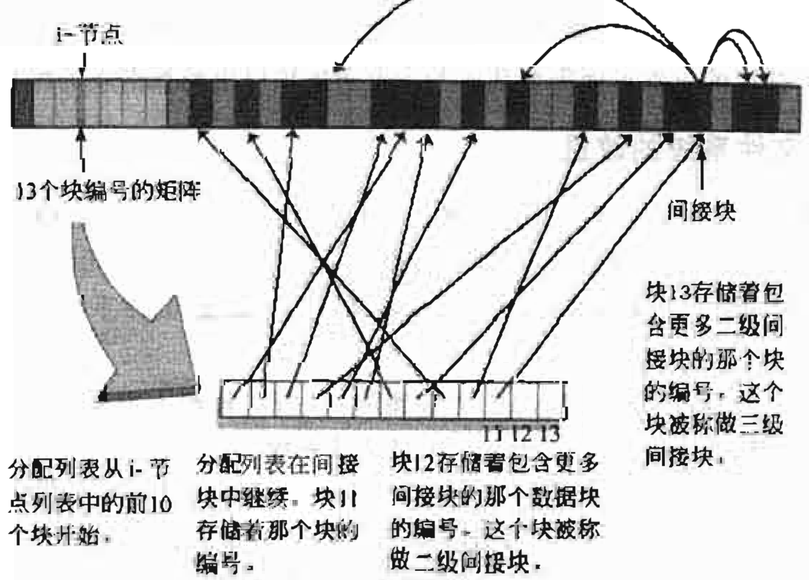


图 4.7　在数据区延伸的块分配列表

"架子空间"——在那 10 个项中的块编号指向的是文件的实际数据。如果分配链表有多于 10 个的项，则剩下的块编号不是存储在 i-节点，而是存储在数据区。存放多余编号的数据块的编号存储在 i-节点的第 11 个项中——就像书店里有个便条贴在架子上写着"剩下的货物在仓库的 3 号架子上"。

注意到这个文件实际上用了 15 个数据块。其中 14 个用来存储文件的内容，剩下的 1 个存放着 i-节点的分配链表无法存放的那部分内容。这个额外的块被称为间接块。

(1) 间接块饱和时如何处理?

当越来越多的字节被添加至文件，内核将分配更多的数据块，因此分配列表越来越长，需要更多的存储空间。分配列表迟早会充满间接块，所以内核将开始引入第二个额外块。内核将如何处理第二个额外块的块编号? 内核需将第二个间接块的编号放入 i-节点的第 12 个项中吗? 可以这么做，但是这样意味着文件仅能含有 3 个额外块。实际上，内核并不把第二个额外块的编号放入 i-节点，取而代之的是，内核开辟另外一个块来存放这些额外间接块的列表。i-节点的第 12 项并不存放第二个额外块的编号，而是存放那个存储着第 2、3、4 及后继额外块的编号的块的编号。这个块被称为二级间接块。

(2) 二级间接块饱和时如何处理?

当二级间接块饱和时，内核开辟另一个新的二级间接块。内核并不把这个新的间接二级块的编号放入 i-节点的列表。而是创建一个三级间接块来存放二级间接块的编号以及这个文件将来所需要的所有间接二级块的编号。i-节点列表的最后一项正是记录着这个三级间接块的编号。

(3) 三级间接块饱和时又将如何处理?

文件到达了极限。如果真想使用大文件，可以创建一个由更大的磁盘块构成的文件系统。当创建该文件系统时，不仅能够定义 i-节点表和数据区的大小，而且能够定义磁盘块的大小。磁盘块并不需要和磁盘扇区同样大小，通常一个磁盘块包含若干个扇区。

由于大文件的存储管理需要更多的开销，因此这样的磁盘分配系统对小文件来说是快捷高效的。当文件逐步增长时，内核使用更多的磁盘空间去维护越来越长的分配列表。在文件中定位一个特定的位置可能需要获取若干个间接块以得到数据块的编号。

4.3.8 Unix 文件系统的改进

前一小节描述了 Unix 文件系统的结构。不同版本的 Unix 使用这种模型的不同版本。这个经典的简洁方法有些严重的不足之处。例如，超级块就是一个问题。如果这个块损坏了，则整个文件系统的结构信息就没有了，新版本的 Unix 在文件系统中备份了这个块的副本。

分块是另一个问题。由于文件的创建和删除，自由块将遍布磁盘。一种方案是在文件系统中创建被称为柱面组(cylinder group)的微文件系统。

但是这个经典的模型并未过时。文件仍旧存储在数据区的块中，文件属性仍旧存储在 i-节点表中的 i-节点中，i-节点包含着磁盘的分配列表；目录仅是文件名和 i-节点号的列表。现在再来回顾一下在前面一章中所创建和探讨的目录树。在理解了文件系统的内部结构以后，再看目录和文件的结构就很清楚了。

4.4 理 解 目 录

在了解了一个 Unix 文件系统的目录结构之后，就能够知道目录树究竟是怎么回事，并且能够理解不同的目录命令是如何工作的。

4.4.1 理解目录结构

用户看到的文件系统是目录和子目录的集合。每个目录能够包含文件和子目录，每个子目录有一个父目录，这棵树的结构常用线条连接的方块图来表示。文件在一个目录中是什么意思？专业术语"d1 是 c 的一个子目录"又是什么意思？图上的线条代表什么意思？

在文件系统内部，目录是一个包含文件名与 i-节点对的列表的文件。从用户的角度看到的是一个文件名的列表，而从 Unix 的角度看到的是一个被命名的指针的列表，如图 4.8 所示。

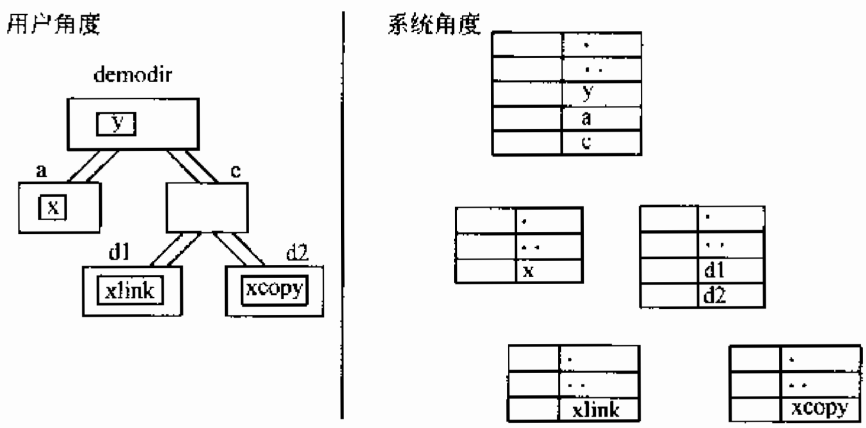


图 4.8　目录树的两种不同视图

如何从用户的角度转换到系统的角度？通过在图中添加 i-节点号，就能够了解目录树是如何连接在一起的。使用 ls -iaR 可以列出一棵树中的所有文件的 i-节点号。

```
$ ls - iaR demodir
865  .    193  ..   277   a    520   c    491   y
demodir/a:
277  .    865  ..   402   x
demodir/c:
520  .    865  ..   651   d1   247   d2
demodir/c/d1:
651  .    520  ..   402   xlink
demodir/c/d2:
247  .    520  ..   680   xcopy
$
```

图 4.9 是上例的图示。

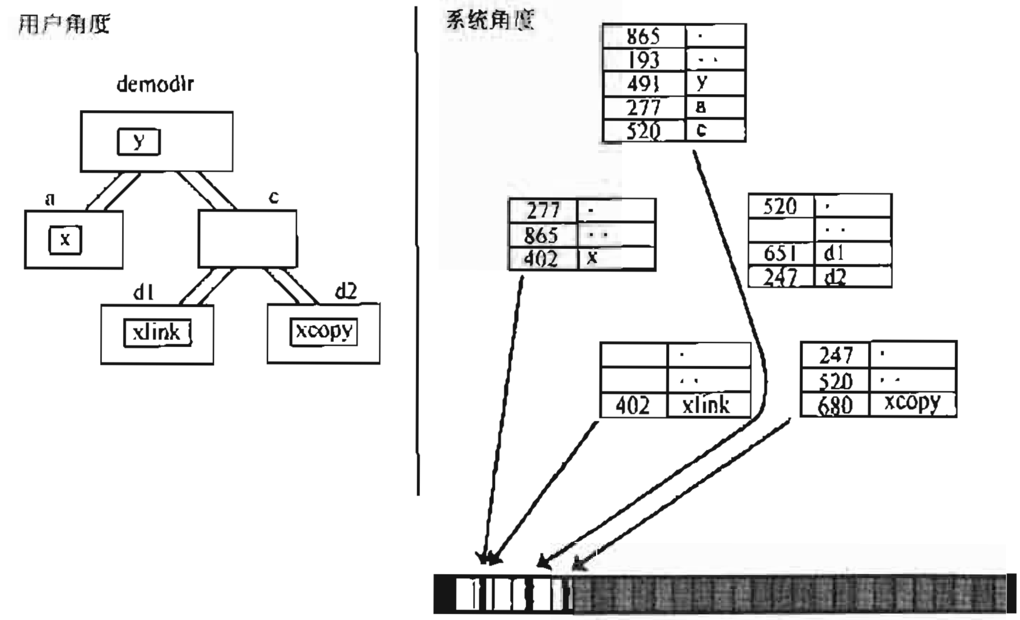


图 4.9　文件名和指向文件的指针

(1)"文件在目录中"的真正含义

一般都说文件存放在某个目录中，但是现在已经知道目录中存放的只是文件在 i-节点表的入口，而文件的内容则存储在数据区。文件在某个目录中是什么意思？例如，从用户的角度来看，文件 y 在目录 demodir 中，而从系统角度来看，看到的则是目录中有一个包含文件名 y 和 i-节点号为 491 的入口。

类似地，"文件 x 在目录 a 中"意味着在目录 a 中有一个指向 i-节点 402 的链接，这个链接所附加的文件名为 x。注意，这一点很重要，在左端较低处标为 d1 的目录包含一个指向 i-节点 402 的链接，那个链接被称为 xlink。称为 demodir/a/x 的链接和称为 demodir/c/d1/xlink 的链接指向同一个文件。

简短地说，目录包含的是文件的引用，每个引用被称为链接。文件的内容存储在数据块，文件的属性被记录在一个被称为 i-节点的结构中，i-节点的编号和文件名存储在目录中。"目录包含子目录"的原理与此相同。

(2)"目录包含子目录"的真正含义

从用户的角度来看，目录 a 是目录 demodir 的一个子目录，那么在系统内部究竟是如何运作的呢？实际上 demodir 包含一个指向那个子目录 i-节点的链接。从系统角度来看，最上面一个表包含一个指向 i-节点 277 的链接，称为 a。如何知道 277 是左边那个目录的 i-节点号呢？每个目录都有一个 i-节点，内核在每个目录都设置一个指向目录本身的 i-节点的入口；这个入口被称为"."。在左边的小方框中，点表示 i-节点 277，因此左边的目录表示 i-节点 277。

如图 4.10 所示，i-节点 520 的目录是如何被包含在 demodir 中的，它在目录 demodir 中的名字被标识为 c。类似地，i-节点 247 是另一个目录，名字为 d2，它是 i-节点 520 的一个

子目录。

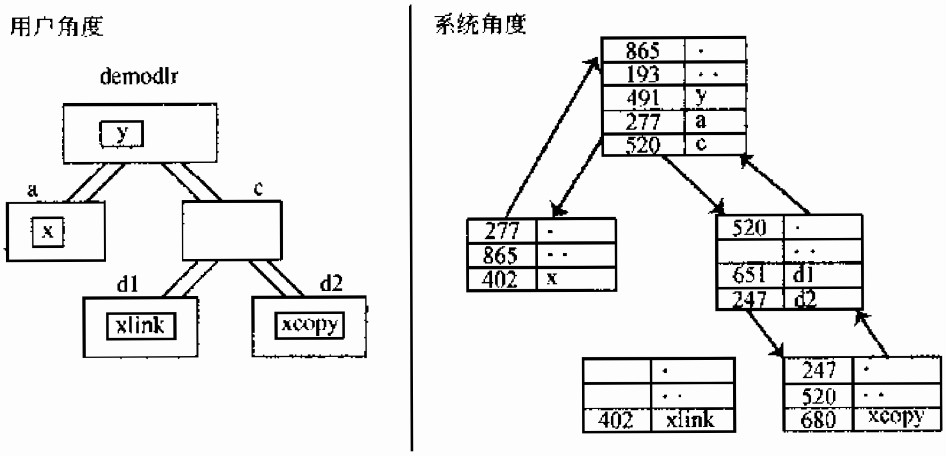


图 4.10　目录名和指向目录的指针

(3) "目录有一个父目录"的真正含义

从用户的角度看目录 d2，它的父目录是 c。这里再次用一个指向 i-节点的简单链接实现，目录 c 的 i-节点号为 520，目录 d2 包含一个称为".."的入口。这个入口的 i-节点号是 520。".."是父目录的保留名字。因此，i-节点 520 是 i-节点 247 的父节点。

(4) 填充空白的 i-节点号

如果理解了前面的章节，接下来就能够填充图 4.10 中的空余的 i-节点号。如果不知道这些空白处该填什么，可以查看一下 ls 的输出内容并复习一下前面章节的内容。

(5) 多重链接及链接数

在 demodir 目录树中，i-节点 402 有两个链接。一个是在目录 a 中，称为 x，另一个在目录 d1 中，称为 xlink。那么哪个是原始文件？哪个是指向它的链接呢？在 Unix 的目录结构中，这两个链接的状态完全相同；它们被称为指向文件的硬链接。文件是一个 i-节点和一些数据块的结合；链接是对 i-节点的引用。可以对一个文件创建任意多的链接。

内核记录了一个文件的链接数。就 i-节点 402 来说，链接数至少是 2。因为在文件系统的其他部分或许还存在着 i-节点 402 的其他链接。链接数被记录在 i-节点中，同时是系统调用 stat 返回值 stat 结构中的一个成员。

(6) 文件名

在 Unix 的文件系统中，文件没有文件名，但是链接具有名字。文件仅仅拥有 i-节点号。在后面的章节中，可看到这种方法的便利之处。

4.4.2　与目录树相关的命令和系统调用

Unix 文件系统的内部结构比较简单，仅仅是一些相互链接的数据结构。节点被称为 i-节点，指针的集合被称为目录，叶子节点被称为链接。通过标准的 Unix 命令来对这个树状结构进行管理，例如 mkdir、rmdir、mv、ln 和 rm 等。这些命令是如何工作的呢？它们将使用哪些相关的系统调用呢？

(1) mkdir

命令 mkdir 用来创建新的目录。它接受命令行上的一个或多个目录名，使用 mkdir 系统调用：

mkdir		
目标	创建目录	
头文件	#include <sys/stat.h> #include <sys/types.h>	
函数原型	int result = mkdir(char * pathname, mode_t mode)	
参数	pathname	新目录名
	mode	权限位的掩码
返回值	−1	遇到错误
	0	成功创建

mkdir 创建一个新的目录节点并把它链接至文件系统树。即 mkdir 创建了这个目录的 i-节点；分配了一个磁盘块用以存储它的内容；在目录中设置两个入口："."和".."，并正确配置了它们的 i-节点号；在它的父目录中增加一个该节点的链接。

(2) rmdir

命令 rmdir 用来删除一个目录。它接受命令行上一个或多个目录名，使用 rmdir 系统调用：

rmdir		
目标	删除一个目录。此目录必须为空	
头文件	#include <unistd.h>	
函数原型	int result = rmdir (const char * path);	
参数	path	目录名
返回值	−1	遇到错误
	0	成功删除

rmdir 从目录树中删除一个目录节点。这个目录必须是空的。即除了"."和".."的入口，这个目录不能包含其他任何的文件和子目录。同时在父目录中删除这个目录的链接。如果这个目录本身并未被其他的进程占用，它的 i-节点和数据块将被释放。

(3) rm

命令 rm 用来从一个目录文件中删除一个记录，它接受命令行上一个或多个文件名，使用 unlink 系统调用：

unlink		
目标	删除一个链接	
头文件	#include <unistd.h>	
函数原型	int result = unlink (const char * path);	
参数	path	需删除的链接名
返回值	−1	遇到错误
	0	成功删除

unlink 用来删除目录文件中的一个记录，减少相应 i-节点的链接数。如果该 i-节点的链接数减为 0，数据块和 i-节点将被释放。如果该 i-节点有其他的链接，则数据块和 i-节点将不受影响。unlink 不能被用来删除目录。

(4) ln

命令 ln 用来创建一个文件的链接，使用系统调用 link：

	link
目标	创建一个文件的新链接
头文件	#include <unistd.h>
函数原型	int result = link (const char * orig, const char * new);
参数	orig 原始链接的名字 new 新建链接的名字
返回值	-1 遇到错误 0 成功创建

link 生成一个 i-节点的链接。新链接包含原始链接的 i-节点号并且具有特定的名字。如果已经存在一个和新链接名相同的链接，则 link 将失败。link 不能被用来生成目录的新链接。

(5) mv

命令 mv 用来改变文件和目录的名字或位置，是这小节中所讲述的最为灵活的一个命令。在很多情况下，mv 仅仅使用系统调用 rename：

	rename
目标	重命名或删除一个链接
头文件	#include <unistd.h>
函数原型	int result = rename (const char * from, const char * to);
参数	from　原始链接的名字 to　新建链接的名字
返回值	-1　遇到错误 0　成功返回

rename 用来改变文件或目录的名字或位置。例如，rename("y", "y.old")用来改变文件的名字，而 rename("y", "c/d2/y.old")用来改变文件的名字和位置。rename 适用于文件和目录，但是在进行目录移动时有些限制。例如，不能将一个目录移动到它的子目录中去。考虑一下 rename("demodir/c", "demodir/d2/c")将会产生怎样的后果。和 link 不同，rename 将删除第一个参数所指定的已存在的文件或空目录。

rename 是如何将一个文件移动到另一个目录的呢？文件实际上并不存在于目录中，目录中存放的仅仅是它的链接。因此，rename 将链接从一个目录移动到另一个目录。将目录 y 移动到 c/d2/y.old 看起来就像图 4.11 所示。

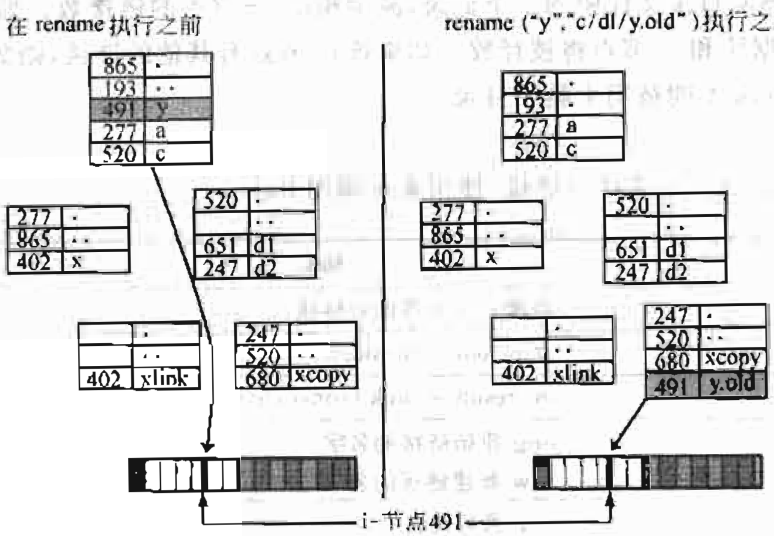


图 4.11　将文件移动到新的目录

首先，在 demodir 中存在着一个指向 i-节点 491 的链接，称为 y。然后，一个称为 y.old 的指向 i-节点的链接出现在 c/d2 中，而原来的链接消失了。内核是如何移动这些链接的呢？

在 Linux 内核，rename 的基本逻辑是：

- 复制链接至新的名字/位置
- 删除原来的链接

Unix 提供系统调用 link 和 unlink 完成这两个操作。因此，rename("x", "z")是这样运作的：

```
if ( link("x", "z") != -1 )
    unlink("x");
```

实际上，过去没有 rename 这个系统调用，因此命令 mv 使用系统调用 link 和 unlink。在内核中增加系统调用 rename 解决了两个问题。首先，rename 使得重命名或重定位一个目录变得更加安全。过去，一般的用户不能对目录进行 link 或 unlink 操作，因此他们没有办法重命名一个目录。

另一个使用 rename 的优点是支持非 Unix 文件系统。在 Unix 中，重命名一个文件或目录是通过改变链接完成的，但是其他的系统可能不按这种方式工作。将通用方法 rename 添加至内核隐藏了实现的细节，使得相同的代码能够在各种文件系统上运行。

(6) cd

cd 用来改变进程的当前目录。cd 对进程产生影响，但是并不影响目录。一个用户可能会说："我进入了/tmp 目录，发现一大堆的垃圾文件"，而另一个用户可能会说："我进入了阁楼，发现了很多旧书"。cd 使用系统调用 chdir。

chdir	
目标	改变所调用进程的当前目录
头文件	#include <unistd.h>
函数原型	int result = chdir (const char * path);
参数	path 要到达的目录
返回值	-1 遇到错误 0 成功改变

Unix 上的每个运行程序都有一个当前目录，chdir 系统调用改变进程的当前目录。在系统内部，进程有一个存放当前目录 i-节点号的变量。从一个目录进入另一个目录只是改变那个变量的值。

了解 cd..如何工作。使用 demodir 这个例子，假设现在位于一个称为 c 的目录，那么，当前目录的 i-节点号是多少？如现在输入 cd d1，则当前目录的 i-节点号是多少？内核是如何获得这个值的呢？如果随后输入 cd ../..，则当前目录的 i-节点号又是多少？内核使用了什么步骤获得该 i-节点号？

如果知道如何完成这个重要的练习，那么已经明白命令 pwd 是如何工作的了。

4.5 编 写 pwd

命令 pwd 用来显示到达当前目录的路径。例如，如果位于 demodir/c/d2 并且输入 pwd，就可以看到：

```
$ pwd
/home/yourname/experiments/demodir/c/d2
```

这么长的路径存放在哪里呢？其实它并不位于当前的目录中。当前目录称呼本身为“.”，并且有一个 i-节点号。目录仅是相互连接的节点集合中的一个节点。pwd 如何知道该目录是 c2，c2 的父目录是 c，c 的父目录是 demodir 呢？

4.5.1 pwd 的工作过程

就像本章中所有问题的答案一样，这个问题的答案也是简单的：追踪链接，读取目录，一个目录接着一个目录地沿着树向上追踪，每步查看“.”的 i-节点号，然后在父目录中查找该 i-节点的名字，直到到达树的顶端。如图 4.12 所示。

从当前目录（就是右下角的那个目录）开始上溯。在该目录当中，所处位置的名称为“.”，拥有 i-节点号 247。现在利用 chdir 向上到达父目录，查找含有 i-节点 247 的入口。在父目录中，i-节点 247 称为 d2。因此，路径的最后一项是 d2。父目录的名字是什么呢？在父目录中，它的名字是“.”，拥有 i-节点号 520。通过 chdir 进入它的父目录，可以看到 i-节点 520 名字为 c。因此，路径的最后两项是 c/d2。算法是以下 3 个步骤的重复。

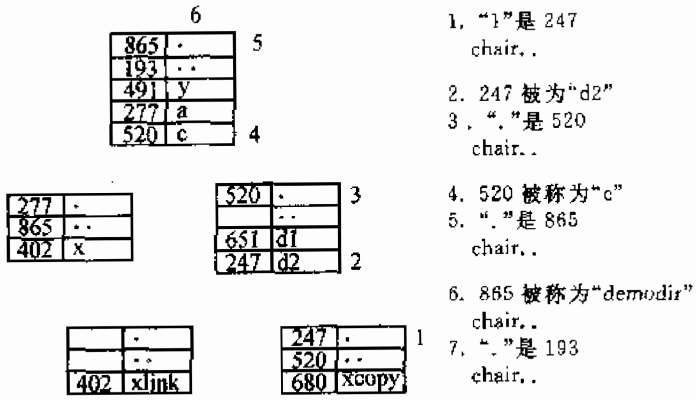


图 4.12　计算当前路径

(1) 得到"."的 i-节点号，称其为 *n*(使用 stat)。

(2) chdir..(使用 chdir)。

(3) 找到 i-节点号 *n* 链接的名字(使用 opendir、readdir、closedir)。

重复(直到到达树的顶端)。

这看起来很简单，但是也有两个问题。

问题 1：如何知道已经到达了树的顶端？在一个 Unix 文件系统的根目录中，"."和".."指向同一个 *i*-节点。编程者通常将下一个指针置为 NULL，用来标识一个链表结构的结束。Unix 的设计者本可以将根目录的".."置为空，但是还是决定将它指向本身。考虑一下这个设计的优点是什么？回到刚才的问题，基于以上原因，pwd 命令重复循环直到一个目录的"."和".."的 i-节点号相同时，就可以认为已经到达文件树的顶端。

问题 2：如何以正确的顺序显示目录名字？可以建立一个循环，使用 strcat 或 sprintf 建立目录名字的字符串序列。通过一个递归的程序逐步到达树的顶端来一个接一个地显示目录名，从而避免了字符串的管理。

4.5.2　pwd 的一种版本

```
/* spwd.c: a simplified version of pwd
 *
 * starts in current directory and recursively
 * climbs up to root of filesystem, prints top part
 * then prints current part
 *
 * uses readdir() to get info about each thing
 *
 * bug: prints an empty string if run from "/"
 * */
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/stat.h>
```

```
#include    <dirent.h>

ino_t get_inode(char * );
void printpathto(ino_t);
void inum_to_name(ino_t , char * , int );

int main()
{
    printpathto( get_inode( "." ) );            /* print path to here */
    putchar('\n');                              /* then add newline */
    return 0;
}

void printpathto( ino_t this_inode )
/*
 * prints path leading down to an object with this inode
 * kindof recursive
 */
{
    ino_t my_inode ;
    char its_name[BUFSIZ];
    if ( get_inode("..") != this_inode )
    {
        chdir( ".." );                                  /* up one dir */
        inum_to_name(this_inode,its_name,BUFSIZ);       /* get its name */
        my_inode = get_inode( "." );                    /* print head */
        printpathto( my_inode );                        /* recursively */
        printf("/%s", its_name );                       /* now print */
                                                        /* name of this */
    }
}
void inum_to_name(ino_t inode_to_find , char *namebuf, int buflen)
/*
 * looks through current directory for a file with this inode
 * number and copies its name into namebuf
 */
{
    DIR     *dir_ptr;                           /* the directory */
    struct dirent *direntp;                     /* each entry */
    dir_ptr = opendir( "." );
    if ( dir_ptr == NULL ){
        perror( "." );
        exit(1);
    }
    /*
     * search directory for a file with specified inum
     */
```

```
    while ( ( direntp = readdir( dir ptr ) ) ! = NULL )
        if ( direntp->d_ino == inode_to_find )
        {
            strncpy( namebuf, direntp->d_name, buflen);
            namebuf[buflen - 1] = '\0'; /* just in case */
            closedir( dir_ptr );
            return;
        }
    fprintf(stderr, "error looking for inum %d\n", inode_to_find);
    exit(1);
}
ino_t get_inode( char * fname )
/*
 * returns inode number of the file
 */
{
    struct stat info;
    if ( stat( fname , &info ) == -1 ){
        fprintf(stderr, "Cannot stat ");
        perror(fname);
        exit(1);
    }
    return info.st_ino;
}
```

下面是命令 pwd 与 spwd 执行后的比较：

```
$ /bin/pwd
/home/bruce/experiments/demodir/c/d2
$ spwd
/ bruce/experiments/demodir/c/d2
$
```

命令 pwd 将显示到达树根的路径，而这个版本在到达树根之前就停止了。问题在哪儿呢？是不是因为编码不当心导致的问题？不是，程序实际上是完全按照原来的设想工作的，它在到达文件系统的根时停止，只是这个文件系统的根并不是这个计算机上整棵文件树的根。

Unix 允许将一个磁盘的存储组织成一棵由多棵树相互连接的树。每个磁盘或磁盘上的每个分区都包含一棵目录树。这些独立的树被连接成一棵单一的几乎无缝的树。这个版本的 pwd 恰好碰上了树之间的连接地带。

4.6　多个文件系统的组合：由多棵树构成的树

一个 Unix 系统有两个磁盘或分区将会如何？现在已经知道，用一些简单的抽象能够将一个单一的分区组织成一棵目录树。但是如果有两个分区，需要两棵独立的树吗？

其他的系统又是如何做的呢？有些操作系统将盘符或卷标分配给每个磁盘或分区，并将字母或名字作为一个文件全路径的一部分。另一种做法是，有些系统统一给所有的磁盘分配块的编号以创建一个虚拟的单一磁盘。

Unix 使用第三种方法。每个分区有自己的文件系统树。当计算机上有多于一个的文件系统时，Unix 提供一种方法将这些树整合成一棵更大的树。图 4.13 表示了这个方法。

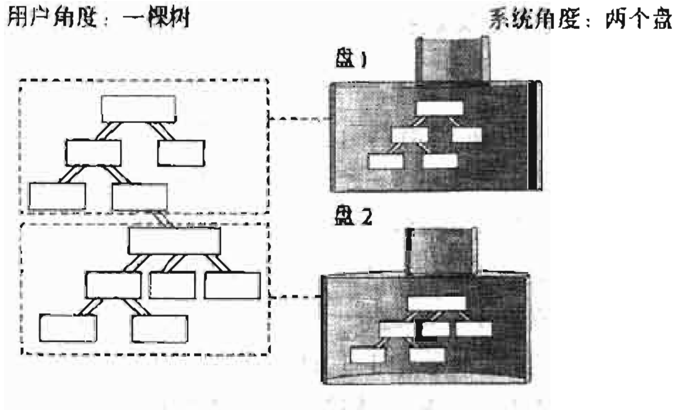


图 4.13　树的嫁接

图 4.13 中用户看到的是一棵完好的目录树，但是实际上有两棵树，一个在磁盘 1 上，一个在磁盘 2 上。每棵树都有一个根目录。一个文件系统被命名为根文件系统，这棵树的顶端是整棵树的真正的根；另一个文件系统则被附加到根文件系统的某个子目录上。在内部，内核在根文件系统将一个目录作为指针，指向另一个文件系统的根，这样两个文件系统就联系起来了。

4.6.1　装载点

在 Unix 中，装载文件系统（to mount a file system）是指将它嵌入到已有的系统以获得某些支持，子树的根目录被嵌入到根文件系统的一个目录中，子树所在的目录被称做第二个系统的装载点（mount point）。

命令 mount 列出当前所装载的文件系统以及它们的装载点：

```
$ mount
/dev/hda1 on / type ext2 (rw)
/dev/hda6 on /home type ext2 (rw)
none on /proc type proc (rw)
none on /dev/pts type devpts (rw, mode = 0620)
$
```

输出的第一行表明/dev/hda 上的分区 1（第一个 IDE 设备）被装载在树的根目录上。这个分区是根文件系统。输出的第二行表明 dev/hda6 上的文件系统被装载在根文件系统的/home 目录上。因此，当用户使用 chdir 从“/”进入“/home”时，实际上是从一个文件系统进入了另一个文件系统。当该 pwd 沿树向上回溯时，它就会停在/home，因为它到达了该文件系统的顶端。

Unix 允许不同类型的文件系统被装载在根文件系统。例如，一个含有 ISO 9660 文件系统的 CD-ROM 能够被装载在一个 Unix 机器上，并且盘上的目录和文件将会成为树的一部分。如果内核包含知道如何与 Macintosh 文件系统协同工作的驱动程序，那么一个含有 Macintosh 文件系统的磁盘就能够被装载。甚至，通过网络连接的其他的计算机上的文件系统也能够被装载。

4.6.2　多重 i-节点号和设备交叉链接

将不同的文件系统合成一棵树有很多优点，当然也有小问题。在 Unix 中，每个文件都有一个 i-节点号。就像两条不同的街道都有一个门牌号为 402 的房子一样，两个不同的磁盘可能都含有 i-节点号为 402 的文件。若干个目录可能都含有 i-节点号为 402 的文件，内核是如何知道应该使用哪个 402 的 i-节点呢？

仔细观察图 4.14 中被圈出来的两个目录，一个在根文件系统，一个在被装载的文件系统。每个目录都包含一个 i-节点 402 的链接。myls.c 和 y.old 似乎为指向同一个 i-节点的两个链接，但是那个 i-节点在哪呢？磁盘 1 上的文件系统有一个 i-节点 402，磁盘 2 上的文件系统有一个不同的 i-节点 402。这两个链接根本不指向同一个文件。

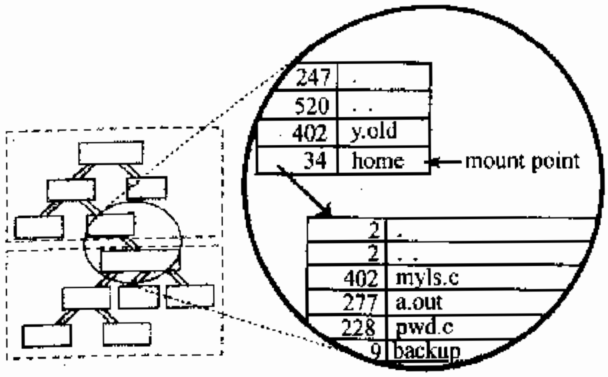


图 4.14　i-节点号和文件系统

这个例子列举了一个由树连接成树的一个问题，即一个 i-节点号并不惟一地标识一个文件了。就像刚刚看到的那样，相同的 i-节点号 402 出现在两个不同的目录中，却指向 2 个不同的文件。看起来这两个链接指向同一个文件，但实际上却不是。

如何从不同的文件系统生成指向同一个文件的链接？这一点是无法做到的，文件以数据块的集合和一个 i-节点的形式出现在磁盘上，目录中的链接会指向那个 i-节点。如果一个磁盘上的链接指向另一个磁盘上的 i-节点将会发生什么事？如果另一个磁盘未被装载，文件则会不存在。更糟糕的是，如果一个存储着拥有 i-节点 402 的不同文件的不同磁盘被装载，则文件的内容就完全不同了。你也可以想到其他麻烦的情况。

link 和 rename 系统调用知道这种情况吗？知道。link 拒绝创建跨越设备的链接，rename 拒绝在不同的文件系统间进行 i-节点号的转移。阅读手册可以了解它们所返回的错误代码。

4.6.3 符号链接

硬链接(hard links)是将目录链接到树的指针，硬链接同时也是将文件名和文件本身链接起来的指针。

硬链接不能指向其他系统中的 i-节点，即使根也不能生成到目录的链接。也许有支持这种跨系统链接的理由，但 Unix 支持另一种形式的链接：符号链接。符号链接通过名字引用文件，而不是 i-节点号。以下是它们的比较：

```
$ who > whoson
$ ln whoson ulist
$ ls -li whoson ulist
377  -rw-r--r--   2  bruce  users  235  Jul  16  09:42  ulist
377  -rw-r--r--   2  bruce  users  235  Jul  16  09:42  whoson
$  ln  -s whoson users
$  ls  -li whoson ulist users
377  -rw-r-r--   2  bruce  users  235  Jul  16  09:42  ulist
289  lrwxrwxrwx  1  bruce  users    6  Jul  16  09:43  users -> whoson
377  -rw-r-r--   2  bruce  users  235  Jul  16  09:42  Whoson
```

文件 whoson 和 ulist 是指向同一个文件的链接。两个都拥有 i-节点号 377，都有同样的文件大小、修改时间和链接数。通过命令 ln 创建硬链接 ulist。

另一方面，命令 ln -s 生成文件 whoson 的一个符号链接，并把这个新链接称为 users。ls -li 显示 users 拥有 i-节点 289。字母 l 在文件类型点处表明 users 是一个符号链接。链接数、修改时间和文件大小都不同于原始文件。文件 users 并不是原始文件 whoson，但是当程序对它进行读写时，它就像原始文件一样。例如：

```
$ wc  -l whoson users
   5 whoson
   5 users
  10 total
$ diff whoson users
$
```

命令 wc 和 diff 分别用于对文件计算行数和比较内容。在这种情况下，内核使用名字找到原始文件。另一方面，对函数 stat 的调用返回关于链接的信息，而不是关于原始文件[①]的。

符号链接可能跨越文件系统，因为它们并不存储原始文件的 i-节点。符号链接也可以指向目录，因为它们与将文件系统联系在一起的真正的链接不同。

在交叉设备链接的情况中符号链接也存在前面所讨论的问题，如果保存原始文件的文件系统被删除了，或者原始文件有了新的文件名，符号链接都将指向空。如果一个拥有相同

① 系统调用 lstat 返回链接所指向的文件的信息。

名字的文件被加载了,符号链接则将指向不同的文件。指向目录的符号链接可以指向父目录,因此在目录树中产生循环套。符号链接可以将你的文件系统彻底搞乱,但是内核知道这些仅是符号链接,而不是真正的链接,所以能够检查丢失的引用和死循环。

系统调用 symlink 用于创建一个符号链接。系统调用 readlink 用于获取原始文件的名字。lstat 用于获取原始文件的信息。阅读联机帮助可以了解 unlink、link 和符号链接是如何作用的。

小　结

1. 主要内容

- Unix 将存储在磁盘中的数据组织成文件系统。文件系统是文件和目录的集合。目录是名字和指针的列表。目录中的每一个入口指向一个文件或目录。目录包含指向父目录和子目录的入口。
- Unix 文件系统包含 3 个主要部分:超级块、i-节点表和数据区域。文件内容存储在数据块。文件属性存储在 i-节点。表中 i-节点的位置称为文件的 i-节点号。i-节点号是文件的惟一标识。
- 相同的 i-节点号可能以不同的名字在若干个目录中出现。每个入口被称为指向文件的硬链接。符号链接是通过文件名引用文件,而不是 i-节点号。
- 若干个文件系统的目录树可被整合成一棵树。内核将一个文件系统的目录链接到另一个文件系统的根的操作称为装载。
- Unix 包含若干种系统调用,允许程序员进行创建和删除目录、复制指针、删除指针、改变连接和分离其他文件系统等的操作。

2. 图示

目录入口是文件名和 i-节点号组成的对。i-节点号指向磁盘上的一个结构,该结构包含文件信息和数据块的分配,如图 4.15 所示。

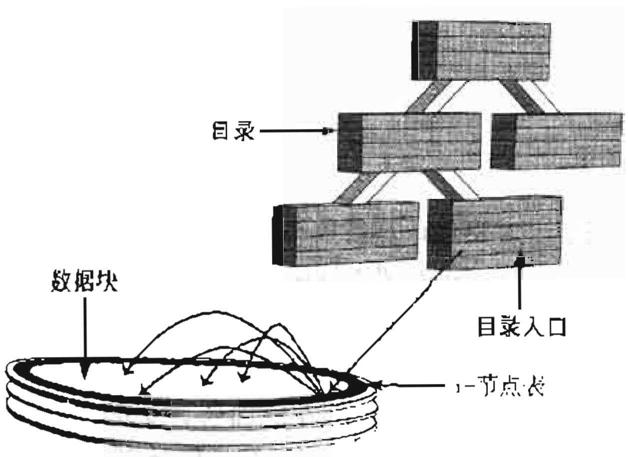


图 4.15　i-节点、数据块、目录、指针

3. 下一章的内容

文件只是一种类型的数据源。程序也可处理来自于像终端、数码相机、扫描仪等设备的数据。Unix 程序如何从设备读取数据和发送数据呢？

4. 习题

4.1　pwd 显示文件系统中到达当前目录的路径。从某种意义上说，那个目录是在树中所处的位置。实际上该目录是一些字节的集合，而这个集合存储在磁盘上的某个位置，该位置能以柱面、磁头、扇区和字节的方式定位。有办法将当前工作目录转换成这些硬件位置吗？

4.2　看一下所使用的系统中的一个硬盘。找出它有多少个分区，确定每个分区的 i-节点的个数和数据块的个数。

4.3　Unix 不仅仅是在内部使用将磁盘抽象成序列的方法创建文件系统，而且它使得这个抽象方法适用于任何拥有合法权限的用户。要做这个实验，需要有最高权限，即管理员权限。

/dev 目录包含允许你从磁盘块中读取字节的设备描述文件，从设备读数据的时候可以认为数据就在这些文件中。在一个装有 IDE 设备的 Linux 系统上，可以看到称做/dev/hda、/dev/hdb、dev/hdc、/dev/hdd 的文件。这些设备文件不是像/etc/passwd 或/var/adm/utmp 这样的常规数据文件。这些数据文件提供对磁盘上原始数据的访问，而且可以使用 cat、more、cp 和其他的命令来读取磁盘上的内容。就像文件 utmp 一样，磁盘有一个清晰的结构。一块接一块地查看磁盘内容的一种方法是使用命令 od -c /dev/hda | more。当查看该命令的输出时，就会觉得磁盘是一个顺序的、连续的磁盘块一样。每个分区都由其中的一个特定文件表示。例如，/dev/hda 上的第一个分区被称为/dev/hda1。

查看系统中的/dev 目录，找出对应于系统上硬盘驱动、软盘驱动、CD-ROM 驱动和其他磁盘驱动的特定文件。

4.4　本章中提到内核在新建一个文件时需要找到一个空的 i-节点和空的磁盘块。内核如何知道哪些磁盘块是空的？哪些 i-节点是空的？你机器上的文件系统使用什么方法跟踪那些未被使用的磁盘块和 i-节点呢？

4.5　Unix 能够读取和装载包含非 Unix 文件系统的磁盘，像 PC-DOS 和 Macintosh 磁盘。在内部，这些系统没有 i-节点。虽然如此，当使用命令 mount 将其中一个磁盘连接到 Unix 系统时，命令 ls -i 将会列出 i-节点。

查看 Linux 源代码以确定这些编号从何处来。Linux 为何添加它们？

4.6　本章中通过描述包含 10 个直接块、1 个间接块、1 个二级间接块和 1 个三级间接块的 i-节点解释了分配列表。有些 Unix 版本使用不同的编号。

(1) 你所使用的系统上的 i-节点分配列表的格式是怎样的？头文件应该包含这些详细内容。

(2) 你系统上的一个数据块的大小是多少?

(3) 在你的系统上,不使用间接块的最大文件是什么?

(4) 在你的系统上,不使用二级间接块的最大文件是什么? 最大的文件实际上用了多少个块?

4.7　一个文件可能有多个链接。链接计数器记录了文件的链接个数。那么目录如何呢? 在你自己的 demodir 树中,使用 ls -l 找出每个目录的链接数。将这些链接数和图表上的箭头相比较。解释目录链接数的意思。为什么每个目录的链接数至少为 2?

4.8　没有人能够使用 link 生成到目录的新链接。在过去,超级用户有权生成到目录的硬链接。在 demodir 的例子中,观测在用户视图和系统视图中添加系统调用 link("demodir/c", "demodir/d2/e")执行后所产生的效果。然后解释命令 ls -iaR demodir 将产生什么结果。

4.9　当使用 mount 命令将一个文件系统装载到另一个文件系统,装载点必须是原有文件系统的一个目录。考虑将/dev/hda4 上的文件系统连接到/home2 这个目录上的情况,思考以下两个问题:(1)如果/home2 这个装载点不存在将产生什么结果? (2)如果装载点存在,且包含文件和子目录,将产生什么结果?

4.10　命令 rmdir 不删除含有文件或子目录的目录。为什么要这么做?
另一方面,可以删除含有用户的目录。尝试以下的操作:生成一个由自己命名的新目录并进入这个目录,然后开启另一个命令窗口,删除这个新目录。关闭第二个命令窗口,输入命令/bin/pwd,看看将产生什么。

4.11　硬盘上的柱面(cylinder)是什么意思? 硬盘的物理构造是什么使得柱面这个概念对有效利用磁盘如此重要? 在网上查找柱面组(cylinder group)这个概念。解释这个概念和本章中文件系统模型之间的联系。

4.12　很多人都了解磁盘空间不足这个概念。一个 Unix 文件系统有一个 i-节点区域和一个数据区域。因此,即使数据区有空间,i-节点空间也有可能不足。当在 Unix 上安装了一个新的磁盘,需要将磁盘分成 i-节点表和数据区。文件系统上的每个文件都需要一个 i-节点。i-节点表越大,留给文件内容的空间越小。
假设要安装一个新的硬盘。命令 mkfs 生成一个新的文件系统并且让你确定 i-节点表的大小。阅读联机帮助以了解这个命令。为什么需要大量的 i-节点? 为什么有时候又比常规需要的少?

4.13　系统调用 stat 接受文件名和指向结构的指针,并且将文件信息填充到该结构中。解释 stat 是如何通过使用目录、i-节点和数据模型来完成该功能的。它是从哪里找到数据并将数据复制到 stat 结构中的呢?

5. 编程练习

4.14 编写一个创建整个 demodir 目录树的 Unix 命令。

4.15 Unix 命令 mkdir 接受选项 -p。编写一种支持这个选项的 mkdir 命令版本。

4.16 命令 mv 不仅仅调用系统调用 rename。编写一种接受两个参数的 mv 版本。第一个参数必须是文件名，第二个参数是文件名或目录名。如果目标是个目录名，则 mv 将文件移动到那个目录。否则，如果可能的话，mv 将重命名这个文件。

4.17 本章中提供了一种使用 link 和 unlink 编写的 rename 版本。该代码片断检验 link 的返回值，但是并未检验 unlink 的返回值。扩充该段代码，使得它能够正确处理 unlink 的错误信息。

4.18 阅读联机帮助和头文件以了解系统上的超级块的结构。编写一个能够打开文件系统、阅读超级块和以清晰可读的格式显示文件系统设置的程序。这个练习与显示 utmp 记录内容和 stat 结构的程序类似。

4.19 对新建一个文件的解释列出了 4 个主要的操作。4 个操作必须完全正确才能使文件能够被正确添加到文件系统。如果计算机在这一系列的操作中突然掉电将会发生什么情况？例如，如果数据被存储在了数据区，而 i-节点还未被分配，将发生什么事？

(1) 为这 4 个操作选择一种顺序，并加以解释。

(2) 现在假设一个系统按照(1)的答案被创建，如果系统在这个过程中突然崩溃将怎么办？例如，你的过程有 4 个步骤、3 个中间点，在每个点的崩溃将会导致文件系统的哪些不一致性呢？

(3) 阅读 Unix 命令 fsck。看看(2)的答案和 fsck 寻找的内容有多少是接近的。

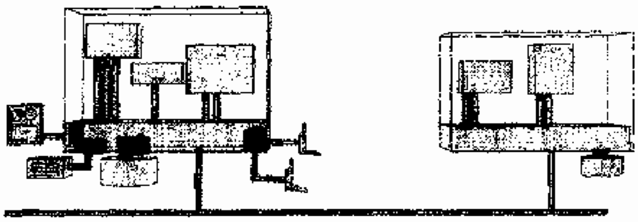
4.20 在第 3 章中编写了 ls -l 的一个版本。修改那个程序，使得它不仅能够显示原先的信息，还能显示 i-节点号。另外，你的新版本的 ls 是从哪里找到 i-节点号的？

6. 项目

基于本章的内容，你能够了解和编写以下这些 Unix 程序：

find、du、ls -R、mount、dump

第 5 章　连接控制：学习 stty



概念与技巧

- 文件和设备间的相似之处
- 文件和设备间的不同之处
- 连接的属性
- 竞争和原子操作
- 控制设备驱动程序
- 流

相关的系统调用

- fcntl、ioctl
- tcsetattr、tcgetattr

相关命令

- stty
- write

5.1　为设备编程

前面章节中已经讲述了一些与文件和目录相关的程序。计算机还有其他的数据来源，如调制解调器、打印机、扫描仪、鼠标、扬声器、照相机和终端等这样的外部设备。在本章中将学习这些设备与目录和文件的相似之处和不同之处，并了解如何将这些想法用于管理设备间的连接。

本章的项目是编写命令 stty 的另外一个版本。stty 用来让用户检测、修改控制键盘和显示器连接属性的设置。

5.2　设备就像文件

很多人认为文件是一些存储在磁盘上的数据，但是 Unix 采用一种更抽象的方法。首先考虑文件的实际情形：文件包含数据，具有属性，通过目录中的名字被标识。可以从一个文

件读取数据，也可以向一个文件写入数据。现在请注意，这种方法将被应用于设备。

考虑一块连接到麦克风和扬声器的声卡。你对着麦克风说话，声卡将来自你声音的信号转换成数据流，使得程序能够读取这个数据流。当程序向声卡写入数据流时，声音就从扬声器中出来。对一个程序来说，声卡既是数据的源，又是数据的目的地。

一个带有键盘和显示器的终端也和文件类似。键盘输入就像数据一样能够被程序读取，而一个进程把写入终端的字符显示在屏幕上。

对 Unix 来说，声卡、终端、鼠标和磁盘文件是同一种对象。在 Unix 系统中，每个设备都被当做一个文件。每个设备都有一个文件名、一个 i-节点号、一个文件所有者、一个权限位的集合和最近修改时间。你所了解的和文件有关的所有内容都将被运用于终端和其他的设备。

5.2.1　设备具有文件名

每个加载到 Unix 机器的设备（终端、打印机、鼠标、磁盘等）都通过文件名表示。通常，表示设备的文件存放在目录/dev 中，但是可以在任何目录中创建设备文件。请查看不同 Unix 机器上的/dev 目录。以下是我所使用的机器上的部分列表：

```
$ ls -C /dev | head -5
XOR        fdlu720    loop1    ptyqf    sda7    stderr    ttysd
agpgart    fdlu800    lp0      ptyr0    sda8    stdin     ttysc
apm_bios   fdlu820    lp1      ptyr1    sda9    stdout    ttysf
arcd       fdlu830    lp2      ptyr2    sdb     tape      ttyt0
dsp        flash0     mcd      ptyr3    sdb1    tcp       ttyt1
```

这个列表显示了若干种设备。第三列中的 lp* 文件是打印机。第二列中的 fd* 文件是软驱。sd* 文件是 SCSI 设备的分区，/dev/tape 是磁带备份驱动程序的设备文件。最后一列中的 tty* 文件是终端。程序通过读取这些文件获得用户的键盘输入，通过写入这些文件向终端屏幕发送数据。

dsp 文件是到声卡的一个连接。进程通过向该设备文件写入字节来运行一个声音文件。进程可以通过打开文件/dev/mouse 来读取鼠标的单击和位置的变化。

5.2.2　设备和系统调用

设备不仅具有文件名，而且支持与所有文件相关的系统调用：open、read、write、lseek、close 和 stat。

例如，从磁带读取数据的代码如下：

```
int fd;
fd = open ("/dev/tape",O_RDONLY);       /* connect to tape drive */
lseek (fd, (long)4096, SEEK_SET);       /* fast forward 4096 bytes*/
n = read (fd, buf, buflen);             /* read data from tape */
close (fd);                             /* disconnect */
```

和磁盘文件相关的系统调用同样可以为其他设备服务。实际上，Unix 没有其他的方法用来和设备通信。

当你移动鼠标并按键，鼠标将数据发送到系统，使得进程能够读取它们。向设备写入数据意味着什么呢？发送数据到鼠标，不会使鼠标移动，也不会使鼠标的键被按下。/dev/mouse文件不支持所有的 write 系统调用。当然，可以制造带有发动机的鼠标，然后编写一个更高级的鼠标驱动程序，使得系统能够接受并产生鼠标事件。

终端支持 read 和 write，但不支持 lseek。考虑一下这是为什么呢？

5.2.3 例子：终端就像文件

Unix 的很多用户输入来自终端。ttysd、ttyse 等文件都代表终端。按传统定义终端是键盘和显示单元，但实际可能包括一个 20 世纪 70 年代生产的打印机、一个键盘和一个串行接口的显示器，或是一个调制解调器和通过拨号上网的软件。在因特网登录的 telnet 或 ssh 窗口也可以认为是一个终端。终端最重要的功能是接受来自用户的字符输入和将输出信息显示给用户。显示输出单元甚至可以产生盲文打印或声音。

命令 tty 用来告知用户所在终端的文件名。用终端文件做以下试验：

```
$ tty
/dev/pts/2
$ cp /etc/motd /dev/pts/2
Today is Monday, we are running low on disk space. Please delete files.
  - your sysadmin
$ who > /dev/pts/2
bruce    pts/2    Jul 17 23:35 (ice.northpole.org)
bruce    pts/3    Jul 18 02:03 (snow.northpole.org)
$ ls -li /dev/pts/2
    4  crw--w--w-  1 bruce tty 136, 2 Jul 18 03:25 /dev/pts/2
```

从以上输出可以知道，终端 tty 对应的设备描述文件名为/dev/pts/2。可以对该文件使用任何与文件相关的命令和进行任何文件操作，如 cp、重定向符“>”、mv、ln、rm、cat 或 ls 等各种命令。

命令 cp 从普通文件/etc/motd 中读取数据，向设备文件/dev/pts/2 写入数据，使得内容能够显示在屏幕上。写入设备文件就是向设备写入字节，例子中的下一行表明将带有重定向符“>”的 who 的输出内容发送到/dev/pts/2，并将数据以字符的形式显示在屏幕上[①]。

5.2.4 设备文件的属性

设备文件具有磁盘文件的大部分属性。上面 ls 的输出内容表明/dev/pts/2 拥有 i-节点 4，权限位为 rw--w--w- ，1 个链接，文件所有者 bruce 和组 tty，最近修改时间是 Jul 18 at

① 这有点像盲人用的点字法。

03:25。文件类型是"c"，表示这个文件实际上是以字符为单位进行传送的设备。权限位看起来有点奇怪，表达式 136,2 显示在表示文件大小的地方，它有什么特殊的含义呢？

(1) 设备文件和文件大小

常用的磁盘文件由字节组成，磁盘文件中的字节数就是文件的大小。设备文件是链接，而不是容器。键盘和鼠标不存储击键数和点击数。设备文件的 i-节点存储的是指向内核子程序的指针，而不是文件的大小和存储列表。内核中传输设备数据的子程序被称为设备驱动程序。

在/dev/pts/2 这个例子中，从终端进行数据传输的代码是在设备一进程表中编号为 136 的子程序。该子程序接受一个整型参数。在/dev/pts/2 中，参数是 2。136 和 2 这两个数被称为设备的主设备号和从设备号。主设备号确定处理该设备实际的子程序，而从设备号被作为参数传输到该子程序。

(2) 设备文件和权限位

每个文件都有相应的读、写和执行的权限。当文件实际上表示设备时，权限位表示什么意思呢？向文件写入数据就是把数据发送到设备，因此，权限写意味着允许向设备发送数据。在这个例子中，文件所有者和组 tty 的成员拥有写设备的权限，但是只有文件的所有者有读取设备的权限。读取设备文件就像读取普通文件一样，从文件获得数据。如果除了文件所有者还有其他用户能够读取/dev/pts/2，那么其他人也能够读取在该键盘上输入的字符，读取其他人的终端输入会引起某些麻烦。

另一方面，向其他人的终端写入字符是 Unix 中 write 命令的目标。

5.2.5 编写 write 程序

在即时消息和聊天室出现之前，Unix 用户通过使用命令 write 和在其他终端上的用户聊天：

```
$ man 1 write
WRITE(1)    Linux Programmer's Mannual    WRITE(1)
Name
    write - send a message to another user
SYNOPSIS
    write user [ttyname]
DESCRIPTION
    Write allows you to communicate with other users by copying lines from you terminal to
    theirs.
    When you run the write command, the user you are writing to gets a message of the form:
        Message from yourname@yourhost on yourtty at hh:mm
        ...
    Any further lines you enter will be copied to the specified user's terminal. If the other user
    wants to reply, they must run write as well.
    When you are done, type an end-of-file or interrupt character. The other user will see the
    message EOF indicating that the conversation is over.
```

以下这个简单的 write 版本仅发送消息内容，而不发送"Message from..."这些提示信息，并且需要的参数是终端的文件名(ttyname)，而不是其他人的用户名：

```
/* write0.c
 *
 * purpose: send messages to another terminal
 *  method: open the other terminal for output then
 *      copy from stdin to that terminal
 *   shows: a terminal is just a file supporting regular i/o
 *   usage: write0 ttyname
 */
#include    <stdio.h>
#include    <fcntl.h>
main( int ac, char * av[] )
{
    int fd;
    char buf[BUFSIZ];
    /* check args */
    if ( ac != 2 ){
        fprintf(stderr,"usage: write0 ttyname\n");
        exit(1);
    }
    /* open devices */
    fd = open( av[1], O_WRONLY );
    if ( fd == -1 ){
        perror(av[1]); exit(1);
    }
    /* loop until EOF on input */
    while( fgets(buf, BUFSIZ, stdin) != NULL )
        if ( write(fd, buf, strlen(buf)) == -1 )
            break;
    close( fd );
}
```

仔细阅读这段代码，在这里找不到键盘连接到其他用户屏幕所需的特殊特征。这个简单的 write 程序将一个文件的内容一行行地复制到另一个文件。这个例程和前面章节中的例子表明终端就像其他连接到 Unix 机器的设备一样，能够以磁盘文件的方式被处理。

5.2.6 设备文件和 i-节点

这些设备文件是如何工作的呢？Unix 文件系统的 i-节点和数据块是如何支持设备文件这个概念的？图 5.1 显示了它们之间的关系。

目录是文件名和 i-节点号的列表。目录并不能区分哪些文件名代表磁盘文件，哪些文件名代表设备。文件类型的区别体现在 i-节点上。

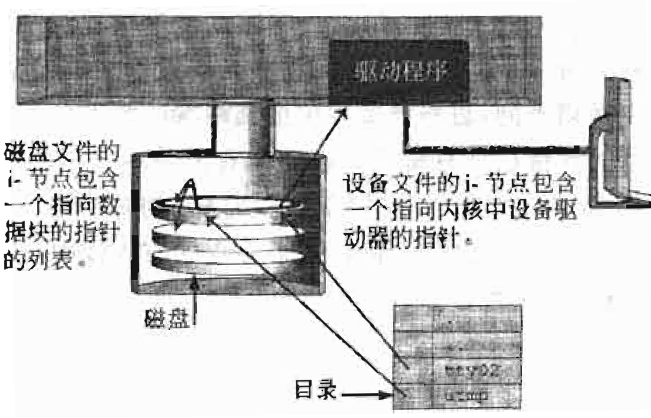


图 5.1　指向数据块或驱动器的 i-节点

每个 i-节点编号指向 i-节点表中的一个结构。i-节点可以是磁盘文件的，也可以是设备文件的。i-节点的类型被记录在结构 stat 的成员变量 st_mode 的类型区域中。

磁盘文件的 i-节点包含指向数据块的指针。设备文件的 i-节点包含指向内核子程序表的指针。主设备号用于告知从设备读取数据的那部分代码的位置。

考虑一下 read 是如何工作的。内核首先找到文件描述符的 i-节点，该 i-节点用于告诉内核文件的类型。如果文件是磁盘文件，那么内核通过访问块分配表来读取数据。如果文件是设备文件，那么内核通过调用该设备驱动程序的 read 部分来读取数据。其他的操作，例如 open、write、lseek 和 close 等都是类似的。

5.3　设备与文件的不同之处

磁盘文件和设备文件都有文件名和属性，从表面上看很类似。系统调用 open 用于创建与文件和设备的连接。但是与磁盘文件的连接不同于与终端的连接。图 5.2 显示了带有两个文件描述符的进程，一个是到磁盘文件的连接，另一个是到终端用户的连接。

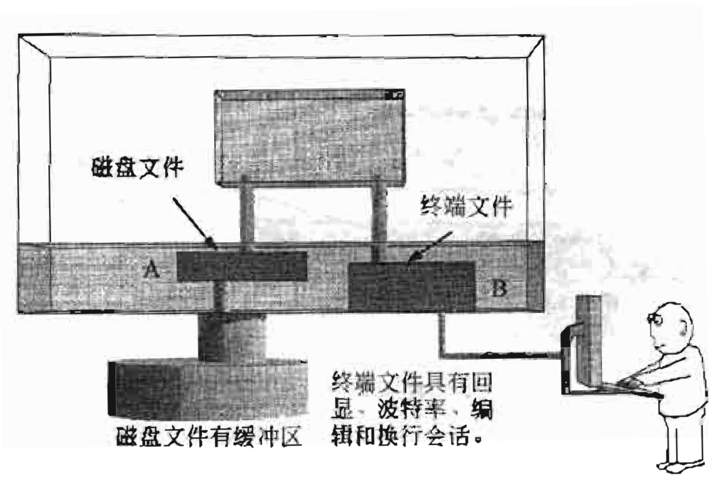


图 5.2　拥有两个文件描述的进程

现在已经了解了一些关于连接的内部情况。与磁盘文件的连接通常包含内核缓冲区。从进程到磁盘的字节先被缓冲，然后才从内核的缓冲区被发送出去。磁盘连接具有缓冲这样一个属性。到终端的连接则不同，进程需要尽快把到终端的数据传送出去。

与终端或调制解调器的连接也具有属性。连接拥有波特率、奇偶位、暂停位的个数。一般情况下所输入的字符都会显示在屏幕上，但是有些时候，例如当输入密码时，字符并不回显在屏幕上。回显字符不是键盘任务的一部分，也不是程序应该做的；回显是连接的一个属性，到磁盘文件的连接没有这些属性。

连接属性和控制

Unix 让文件和设备既有相似之处，又有不同之处。与磁盘文件的连接不同于与调制解调器的连接。关于连接的属性可以问：

1. 连接可有哪些属性？
2. 如何检测当前的属性？
3. 如何改变当前的属性？

下面介绍两例：磁盘连接的属性和终端连接的属性。

5.4 磁盘连接的属性

系统调用 open 用于在进程和磁盘文件之间创建一个连接。该连接含有若干个属性，下面先仔细学习其中的两个属性，然后再了解一下其他的属性。

5.4.1 属性 1：缓冲

图 5.3 显示了当两个管道通过一个进程单元连接时文件描述符的情况。那个进程单元是用来进行缓冲和完成其他进程任务的。在方框内的是控制变量，用以决定文件描述符应该采取哪个进程步骤。

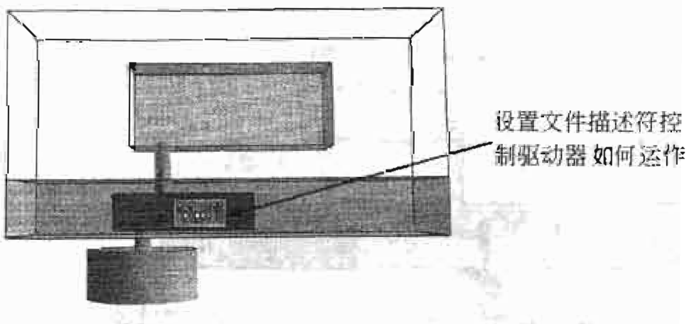


图 5.3 数据流中的进程单元

可以通过修改控制变量改变文件描述符的动作。例如，通过简单的 3 步操作关闭磁盘缓冲，如图 5.4 所示。

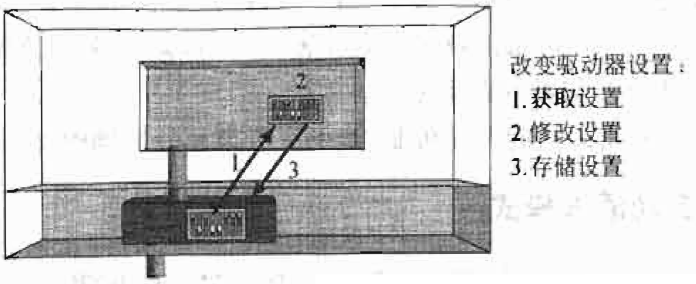


图 5.4　修改文件描述符的运作

首先，生成一个系统调用将控制变量从文件描述符复制到进程。然后，修改这个复制过来的控制变量。最后，将修改过的值送回内核。新的设置被安置在进程代码中，内核根据新的设置处理数据。下面是遵循上述 3 步的代码：

```
#include <fcntl.h>
int s;                                  //settings
s = fcntl (fd, F_GETFL);                //get flags
s |= O_SYNC;                            //set SYNC bit
result = fcntl (fd, F_SETFL, s);        //set flags
if ( result == 1)                       //if error
   perror ("setting SYNC");             //report
```

文件描述符的属性被编码在一个整数的位中。系统调用 fcntl 通过读写该整数位来控制文件描述符。

	fcntl
目标	控制文件描述符
头文件	#include <fcntl.h> #include <unistd.h> #include <sys/types.h>
函数原型	int result = fcntl(int fd, int cmd); int result = fcntl(int fd, int cmd, long arg); int result = fcntl(int fd, int cmd, struct flock * lockp);
参数	fd　需控制的文件描述符 cmd　需进行的操作 arg　操作的参数 lock　锁信息
返回值	−1　遇到错误 other　依操作而定

fcntl 在 fd 所指定的文件上执行操作 cmd。arg 代表操作 cmd 所使用的一个参数。在上例中，参数 F_GETFL 得到当前的位集(也就是 flags)。变量 s 存放这个 flag 集。位逻辑或操作打开位 O_SYNC。该位告诉内核，对 write 的调用仅能在数据写入实际的硬件时才能返回，而不是在数据复制到内核缓冲时就执行默认的返回操作。

最后，把修改过的设置返回内核。将 F_SETFL 操作作为第二个参数，将修改过的设置作为第三个参数。这 3 个步骤(从内核中读取设置到变量，修改这些设置，将设置返回内核)是 Unix 中读取和修改连接属性的典型方法。

设置 O_SYNC 会关闭内核的缓冲机制，如果没有很充分的理由，最好不要关闭缓冲。

5.4.2 属性 2：自动添加模式

文件描述符的另一个属性是自动添加模式(auto-append mode)。自动添加模式对于若干个进程在同一时间写人文件是很有用的。

为什么自动添加是有用的？

考虑日志文件 wtmp。wtmp 存储所有的登录和退出记录。当一个用户登录时，程序 login 在 wtmp 的末尾追加一条登录记录。当一个用户退出时，系统在 wtmp 的末尾追加一条退出的记录，如同系统维护的日记一样。这就像人们写日记一样，每篇都被添加在末尾。

不能使用 lseek 在末尾进行添加记录吗？考虑一下登录的逻辑，如图 5.5 所示。

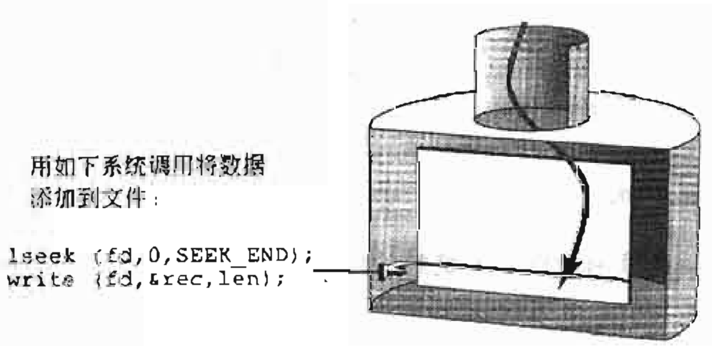


图 5.5　用 lseek 和 write 进行添加

lseek 将当前位置移到文件的末尾，然后添加登录的记录。这里会产生什么错误呢？

如果两个人同时登录将会发生什么？含有时间过程，如图 5.6 所示。

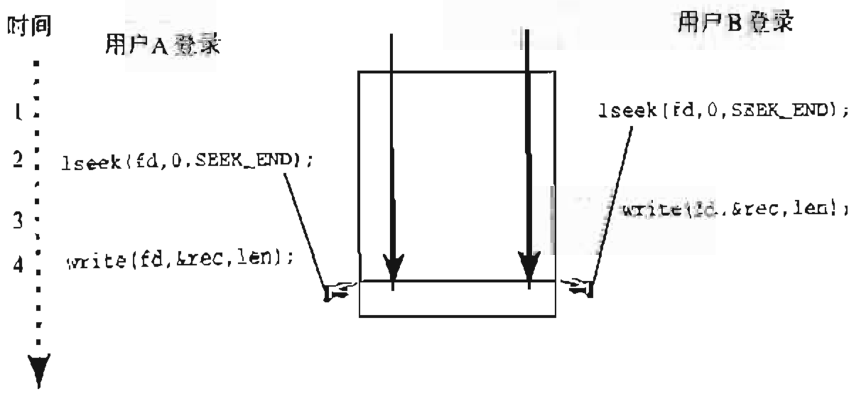


图 5.6　lseek 和 write 所引起的混乱

wtmp 文件显示在中间，时间箭头在左边，并显示了 4 个时间片断。用户 A 登录的代码显示在左边，用户 B 登录的代码显示在右边。到现在为止一切都正常吗？一个重要的事实

是，Unix 是一个时间共享系统，这个过程需要两个独立的步骤：lseek 和 write。

现在仔细看看下面：

- 时间 1—B 的登录进程定位文件的末尾
- 时间 2—B 的时间片用完，A 的登录进程定位文件的末尾
- 时间 3—A 的时间片用完，B 的登录进程写入记录
- 时间 4—B 的时间片用完，A 的登录进程写入记录

因此，A 的登录进程写入的记录覆盖了 B 的记录，B 的登录记录丢失。

这种情况被称为竞争(race condition)。这两个进程所共享的网状效应依赖于这两个进程如何规划。在时间方面做一个小的改动，可能 A 的登录记录会丢失，可能两个都不会丢失。

如何避免这种竞争？有很多方法避免竞争。竞争是系统编程所面临的重要问题，后面需要多次回到这个话题。在这个特定的情况中，内核提供一个简单的解决办法：自动添加模式。当文件描述符的 O_APPEND 位被开启后，每个对 write 的调用自动调用 lseek 将内容添加到文件的末尾。

下面的代码启动自动添加模式，然后调用 write：

```
#include <fcntl.h>
int s;                              //settings
s = fcntl ( fd, F_GETFL);           //get flags
s |= O_APPEND;                      //set APPEND bit
result = fcntl ( fd, F_SETFL, s);   //set flags
if ( result == -1)                  //if error
   perror ( "setting APPEND");      // report
else
   write (fd, &rec, 1);             //write record at end
```

术语竞争和原子操作(atomoc operation)密切相关。对 lseek 和 write 的调用是独立的系统调用，内核可以随时打断进程，从而使后面这两个操作被中断。当 O_APPEND 被置位，内核将 lseek 和 write 组合成一个原子操作，被连接成一个不可分割的单元。

5.4.3　用 open 控制文件描述符

O_SYNC 和 O_APPEND 是文件描述符的两个属性，其他的属性将在后面的章节中讨论。fcntl 的联机帮助列出了你的系统上所支持的所有选项和操作。

fcntl 并不是仅有的用来设置文件描述符属性的方法。通常在打开一个文件时，应该知道需要怎样的设置。可以通过系统调用 open 的第二个参数的一部分来设置文件描述符的属性位。例如，调用：

```
fd = open ( WTMP_FILE, O_WRONLY | O_APPEND | O_SYNC);
```

以写方式打开文件 wtmp 并将 O_APPEND 和 O_SYNC 位开启。open 的第二个参数不只是读、写或读/写的选择。

例如，可以通过 open 创建一个包含 O_CREAT 标志位的文件。以下两个调用是等价的：

```
fd = creat ( filename, permission_bits);
fd = open ( filename, O_CREAT | O_TRUNC | O_WRONLY, permission_bits);
```

为什么 open 可以实现相同的功能，而 creat 依旧存在？在老的版本中，open 仅仅用来打开文件，creat 用来创建新的文件。随后，open 被多次修改以支持更多的标志位，包括创建文件选项。

open 支持的其他标志位：

O_CREAT　如果不存在，创建该文件。可查看 O_EXCL。
O_TRUNC　如果文件存在，将文件长度置为 0。
O_EXCL　O_EXCL 标志位防止两个进程创建同样的文件。如果文件存在且 O_EXCL 被置位，则返回 -1。

O_CREAT 和 O_EXCL 的组合用来消除以下竞争情况：如果两个进程同时创建相同的文件将会发生什么情况？例如，如果两个进程都要写 wtmp，但是这个文件不存在时，都要创建该文件，此时会发生什么情况？程序能够先调用 stat 查看文件是否存在，如果不存在，就调用 creat。当 stat 和 creat 间的过程被打断时，问题就出现了。O_EXCL/O_CREAT 的组合将这两个调用构成了一个原子操作。虽然想法很好，但是这种方法在某些重要场合并不可行。一个可靠的替代方案是使用 link。本章的练习提供了一个例子。

5.4.4　磁盘连接小结

内核在磁盘和进程间传输数据。内核中进行这些传输的代码有很多选项。程序可使用 open 和 fcntl 系统调用控制这些数据传输的内部运作，如图 5.7 所示。

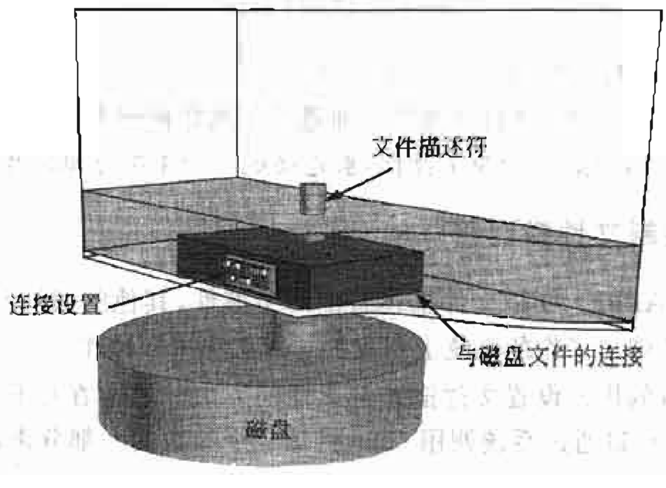


图 5.7　与文件的连接具有属性设置

5.5　终端连接的属性

系统调用 open 在进程和终端之间创建一个连接。现在来仔细了解一下与终端连接的一些属性。

5.5.1　终端的 I/O 并不如此简单

终端和进程之间的连接看起来简单。通过使用 getchar 和 putchar 就能够在设备和进程间传输字节。数据流的这种抽象使得键盘和屏幕看起来就像在进程中一样，如图 5.8 所示。

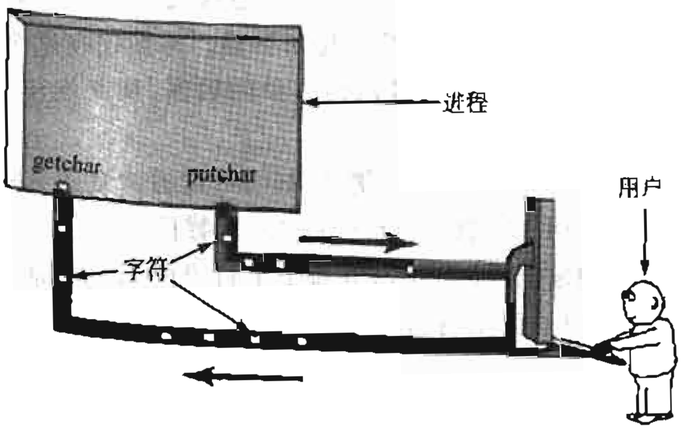


图 5.8　一个简单、直接连接的流程

一个简单的实验表明这个模型并不完整。考虑以下这个程序：

```
/* listchars.c
 * purpose: list individually all the chars seen on input
 *  output: char and ascii code, one pair per line
 *   input: stdin, until the letter Q
 *   notes: usesful to show that buffering/editing exists
 */
#include    <stdio.h>
main()
{
   int c, n = 0;
   while( ( c = getchar()) != 'Q' )
      printf("char %3d is %c code %d\n", n++ , c, c );
}
```

这个程序以一个接一个的方式处理字符，读取字符，打印数值、字符本身以及它的内部代码。编译并运行这个程序，结果如下所示：

```
$ ./listchars
```

```
hello
char    0 is h code 104
char    1 is e code 101
char    2 is l code 108
char    3 is l code 108
char    4 is o code 111
char    5 is
code 10
Q
$
```

接下来会发生什么事情？如果字符代码直接从键盘流向 getchar，则在每个字符后可看到一个响应。输入单词 hello 中的 5 个字符并按回车键。然而仅在这个时候，程序才开始处理这些字符。输入看起来被缓冲了。就像流向磁盘的数据，从终端流出的数据在沿途中的某个地方被存储起来了。

listchars 显示了另外一些内容。Enter 键或 Return 键通常发送 ASCII 码 13，即回车符。listchars 的输出显示 ASCII 码 13 被换行符(代码 10)所替代。

第三种处理影响程序的输出。listchars 在每个字符串的末尾添加一个换行符(\n)。换行符代码告诉鼠标移到下一行，但没有告诉它移到最左边。代码 13(回车符)告诉鼠标回到最左端[1]。

运行 listchars 表明在文件描述符的中间必定有一个处理层。图 5.9 显示了该层的部分作用。

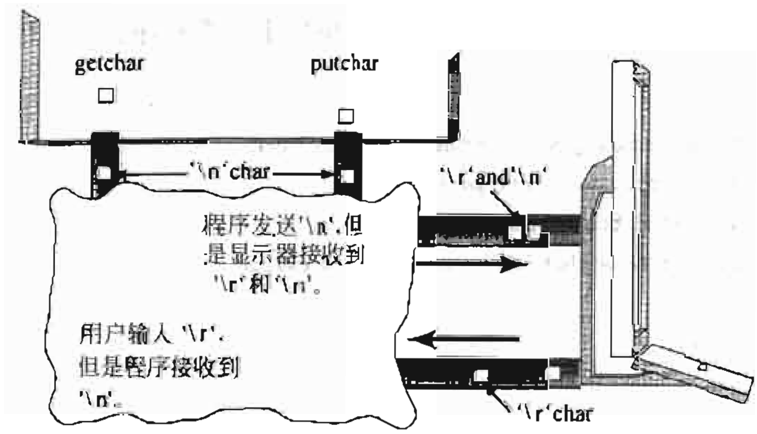


图 5.9 内核处理终端数据

这个例子说明了 3 种处理：

1. 进程在用户输入 Return 后才接收数据；
2. 进程将用户输入的 Return(ASCII 码 13)看作换行符(ASCII 码 10)；
3. 进程发送换行符，终端接收回车换行符。

① 你可以向你的爷爷请教如何在打字机上推左侧手柄使打字头回到纸的左边。

与终端的连接包含一套完整的属性和处理步骤。

5.5.2 终端驱动程序

终端和进程之间的连接如图 5.10 所示。

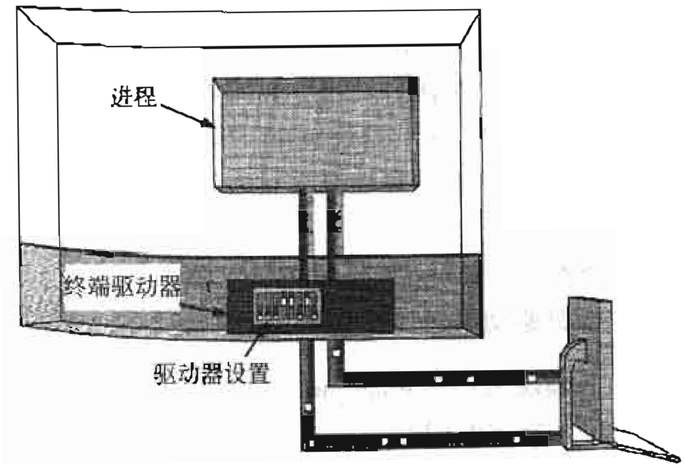


图 5.10 终端驱动器是内核的一部分

处理进程和外部设备间数据流的内核子程序的集合被称为终端驱动程序或 tty 驱动程序[①]。驱动程序包含很多控制设备操作的设置。进程可以读、修改和重置这些驱动控制标志。

5.5.3 stty 命令

stty 命令让用户读取和修改终端驱动程序的设置。

(1) 使用 stty 显示驱动程序设置

stty 的输出如下所示：

```
$ stty
speed 9600 baud; line = 0;
$ stty -all
speed 9600 baud; rows 15; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint ignpar -parmrk -inpck istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
```

① tty 是指由 Teletype 公司生产的老式打印终端。

```
echoctl echoke
```

默认选项的列表很简洁。如加上选项 -all 则将列出更多的设置。有些设置是有值的变量，有些是布尔值。例如，波特率和屏幕的行数与列数拥有数值。像 intr、quit 和 eof 这些项拥有字符值。而像 icrnl、-olcuc 和 onlcr 的值是开或关。

这些意味着什么？icrnl 是 Input:convert Carriage Return to NewLine(输入时将回车转换为换行)的缩写，即在前面的例子中驱动程序所做的操作。缩写 onlcr 代表 Output:add to NewLine a Carriage Return(输出时在新的一行中加入回车)。一个属性前的减号表示这个操作被关闭。例如，-olcuc 表示动作 Output:convert LowerCase to UpperCase(输出时将小写字母转换成大写)被禁止。很多早期的终端只显示大写字母，所以那时将输出转换成大写很有用。

(2) 使用 stty 改变驱动程序设置

这里是一些使用 stty 修改驱动程序属性的例子：

```
$ stty erase X          # make 'X' the erase key
$ stty -echo            # type invisibly
$ stty erase @ echo     # multiple requests
```

在第一个例子中，使用 stty 用来改变删除键。退格键或删除键是典型设置，但是可以将任何键作为删除键①。在第二个例子中，关闭按键回显。当输入密码时，字符并不回显在屏幕上。关闭这个回显意味着能够打字，但是看不到所输入的字符。在第三个例子中，使用 stty 一次性改变多种设置。同时将删除键改为@，并将回显模式开启。

stty 如何运作？现在能够编写 stty 了吗？

5.5.4　编写终端驱动程序：关于设置

tty 驱动程序包含很多对传入的数据所进行的操作。这些操作被分为 4 种：

- 输入：驱动程序如何处理从终端来的字符
- 输出：驱动程序如何处理流向终端的字符
- 控制：字符如何被表示——位的个数、位的奇偶性、停止位等
- 本地：驱动程序如何处理来自驱动程序内部的字符

输入处理包括将小写字母转换为大写字母，去除最高位及将回车符转换为换行符。输出处理包括用若干个空格符代替制表符，将换行符转换为回车符及将小写字母转换为大写字母。控制设置包括奇偶性及停止位的个数。本地处理包括回显字符给用户及缓冲输入直到用户按回车键。

除了开和关设置外，驱动程序维护了一张含有特殊意义键的列表。例如，用户可能按退格键来删除一个字符。终端驱动程序会注意并处理这个删除键。除此之外，终端驱动程序还负责对其他一些控制字符进行处理。

联机帮助上列出了 stty 大部分的设置和控制字符。

① 部分 Unix shell 处理此命令，并提供比驱动程序更强大的功能，而不在驱动程序中处理此命令。

5.5.5 编写终端驱动程序：关于函数

改变终端驱动程序的设置就像改变磁盘文件连接的设置一样：

(1) 从驱动程序获得属性；

(2) 修改所要修改的属性；

(3) 将修改过的属性送回驱动程序。

例如，以下代码为一个连接开启字符回显：

```
# include <termios.h>
struct termios attribs;                    /* struct to hold attributes */
tcgetattr ( fd, &settings);                /* get attribs from driver */
settings.c_lflag | = ECHO;                 /* turn on ECHO bit in flagset */
tcsetattr ( fd, TCSANOW, &settings);       /* send attribs back to driver */
```

通常的过程在图 5.11 中进行描述。

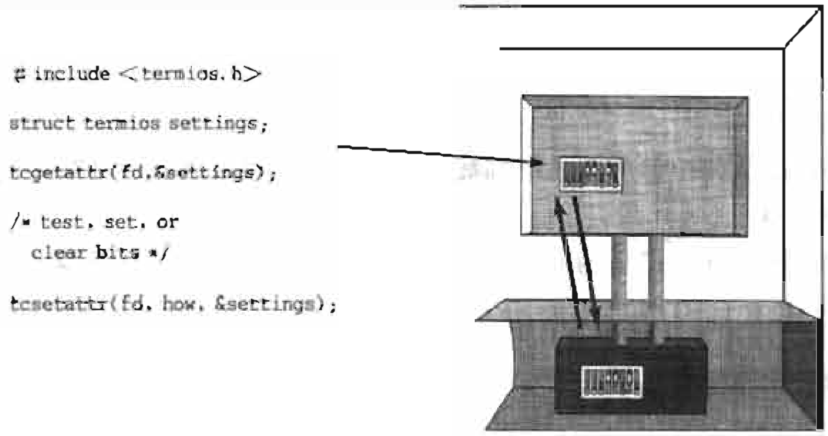


图 5.11　调用 tcgetattr 和 tcsetattr 控制终端驱动器

库函数 tcgetattr 和 tcsetattr 提供对终端驱动程序的访问。两个函数在 termios 结构中交换设置。以下是详细描述。

tcgetattr	
目标	读取 tty 驱动程序的属性
头文件	# include <termios.h> # include <unistd.h>
函数原型	int result = tcgetattr(int fd, struct termios * info);
参数	fd　　与终端相联的文件描述符 info　　指向终端结构的指针
返回值	−1　　遇到错误 0　　成功返回

tcgetattr 从与文件 fd 相关的终端驱动程序中获取当前设置，并把它复制到 info 指针所指的结构中。

tcsetattr		
目的	设置 tty 驱动程序的属性	
头文件	#include <termios.h> #include <unistd.h>	
函数原型	int result = tcsetattr(int fd, int when, struct termios * info);	
参数	fd	与终端相联的文件描述符
	when	改变设置的时间
	info	指向终端结构的指针
返回值	-1	遇到错误
	0	成功返回

tcsetattr 从 info 所指的结构中将驱动程序的设置复制到与文件 fd 相关的终端驱动程序中。when 参数告诉 tcsetattr 在什么时候更新驱动程序设置。when 的允许值如下所示。

(1) TCSANOW

立即更新驱动程序设置。

(2) TCSADRAIN

等待直到驱动程序队列中的所有输出都被传送到终端。然后进行驱动程序的更新。

(3) TCSAFLUSH

等待直到驱动程序队列中的所有输出都被传送出去。然后，释放所有队列中的输入数据，并进行一定的变化。

5.5.6 编写终端驱动程序：关于位

termios 结构类型包括若干个标志集和一个控制字符的数组。所有的 Unix 版本包含以下结构：

```
struct termios
  {
    tcflag_t   c_iflag;      /* input mode flags */
    tcflag_t   c_oflag;      /* output mode flags */
    tcflag_t   c_cflag;      /* control mode flags */
    tcflag_t   c_lflag;      /* local mode flags */
    cc_t       c_cc[NCCS];   /* control characters */
    speed_t    c_ispeed;     /* input speed */
    speed_t    c_ospeed;     /* output speed */
  };
```

从驱动程序进出的数据流的波特率存储在 c_ispeed 和 c_ospeed 成员中。

每个标志集的独立位的含义如图 5.12 所示。

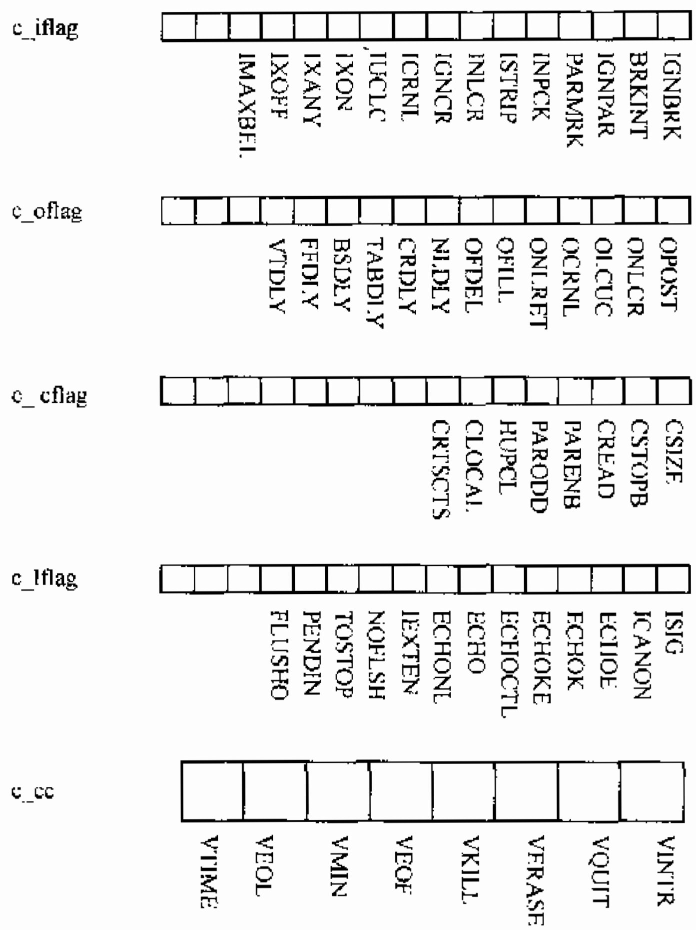


图 5.12　终端变量中的位和字符

首先描述的 4 个成员是标志集。每个标志集包含在该组中的操作位。例如，成员 c_iflag 设置 INLCR 值的位。成员 c_cflag 设置掩码 PARODD 的值，其功能是设置奇偶性。所有这些掩码都定义在 termios.h 中。如从驱动程序中读取当前的属性到 termios 结构中时，这个结构中的所有值都可以被检验和修改。

成员 c_cc 是控制字符的数组。含有特殊功能的键都被存储在这个数组中。数组中的每个位置都由 termios.h 中的常量所定义。例如，attribs.c_cc[VERASE] = '\b'告诉驱动程序将退格键作为删除键。

现在已经知道如何从驱动程序获得设置和如何将设置存回驱动程序，下面看看修改驱动程序属性的技术。

每个属性在标志集中都占有一位。属性的掩码定义在 termios.h 中。要测试一个属性，需要将标志集与那个位的掩码相与。要启动这个属性，将该位开启。要禁止这个属性，将该位关闭。上面的情况如下所示：

操作	代　　码
测试位	if (flagset & MASK) ...
置位	flagset | = MASK
清除位	flagset & = ~MASK

5.5.7　编写终端驱动程序：几个程序例子

1. 例子：echostate.c——显示回显位的状态

第一个例子说明终端是否被设置成回显字符的模式。读取设置，测试位，并报告结果：

```
/* echostate.c
 * reports current state of echo bit in tty driver for fd 0
 * shows how to read attributes from driver and test a bit
 */
#include    <stdio.h>
#include    <termios.h>
main()
{
   struct termios info;
   int rv;

   rv = tcgetattr( 0, &info ); /* read values from driver */
   if ( rv ==  -1 ){
        perror( "tcgetattr");
        exit(1);
   }
   if ( info.c_lflag & ECHO )
        printf(" echo is on , since its bit is 1\n");
   else
        printf(" echo is OFF, since its bit is 0\n");
}
```

这个程序为文件描述符 0 读取终端属性。0 是标准输入的文件描述符，该文件描述符通常附属在键盘上。这里是编译和运行程序的一个例子：

```
$ cc echostate.c -o echostate
$ ./echostate
echo is on, since its bit is 1
$ stty -echo
$ ./echostatr: not found
$ echo is OFF, since its bit is 0
```

这个例子显示命令 stty -echo 关闭驱动器里的击键回显。用户在这之后输入了另外两个命令，但它们在屏幕上并不显示。另一方面，对那两行的输出响应仍然显示。

2. 例子：setecho.c——改变回显位的状态

第二个例子将键盘回显开或关。如果命令行参数以“y”开始，终端的回显标志被开启。否则回显被关闭。程序如下所示：

```
/* setecho.c
 *  usage: setecho [y|n]
 *  shows: how to read, change, reset tty attributes
 */
#include    <stdio.h>
#include    <termios.h>

#define oops(s,x) { perror(s); exit(x); }

main(int ac, char *av[])
{
   struct termios info;
   if ( ac == 1 )
   exit(0);
   if ( tcgetattr(0,&info) == -1 )            /* get attribs */
         oops("tcgettattr", 1);
   if ( av[1][0] == 'y' )
         info.c_lflag |= ECHO ;               /* turn on bit */
   else
         info.c_lflag &= ~ECHO ;              /* turn off bit */

   if ( tcsetattr(0,TCSANOW,&info) == -1 )    /* set attribs */
         oops("tcsetattr",2);
}
```

测试并运行这两个程序以及正常模式下的 stty：

```
$ echostate; setecho n ; echostate ; stty echo
echo is on, since its bit is 1
echo is OFF, since its bit is 0
$ stty -echo; echostate ; setecho y ; setecho n
echo is OFF, since its bit is 0
```

在第一个命令行使用 setecho 关闭回显。然后使用 stty 将回显重新开启。驱动程序和驱动程序设置被存储在内核，而不是在进程。一个进程可以改变驱动程序里的设置，另一个不同的进程可以读取或修改设置。

3. 例子：showtty.c——显示大量驱动程序属性

可以重复用 setecho.c 和 echostate.c 中的技术建立一个完整的 stty 版本。tty 驱动程序包含 3 种设置：特殊字符、数值和位。showtty 包含显示这些数据类型的函数。以下是代码：

```
/* showtty.c
```

```
 * displays some current tty settings
 */

#include    <stdio.h>
#include    <termios.h>

main()
{
    struct termios ttyinfo;                     /* this struct holds tty info */
    if ( tcgetattr( 0 , &ttyinfo ) == -1 ){     /* get info */
        perror( "cannot get params about stdin");
        exit(1);
    }
                                                /* show info */
    showbaud ( cfgetospeed( &ttyinfo ) );       /* get + show baud rate */
    printf ("The erase character is ascii %d, Ctrl - %c\n",
            ttyinfo.c_cc[VERASE], ttyinfo.c_cc[VERASE] - 1 + 'A');
    printf("The line kill character is ascii %d, Ctrl - %c\n",
            ttyinfo.c_cc[VKILL], ttyinfo.c_cc[VKILL] - 1 + 'A');
    show_some_flags( &ttyinfo );                /* show misc. flags */
}
showbaud( int thespeed )
/*
 * prints the speed in english
 */
{
    printf("the baud rate is ");
    switch ( thespeed ){
        case B300:  printf("300\n"); break;
        case B600:  printf("600\n"); break;
        case B1200: printf("1200\n"); break;
        case B1800: printf("1800\n"); break;
        case B2400: printf("2400\n"); break;
        case B4800: printf("4800\n"); break;
        case B9600: printf("9600\n"); break;
        default: printf("Fast\n"); break;
    }
}
struct flaginfo { int fl_value; char *fl_name; };

struct flaginfo input_flags[] = {
        IGNBRK ,    "Ignore break condition",
        BRKINT ,    "Signal interrupt on break",
        IGNPAR ,    "Ignore chars with parity errors",
        PARMRK ,    "Mark parity errors",
```

```
        INPCK ,     "Enable input parity check",
        ISTRIP ,    "Strip character",
        INLCR ,     "Map NL to CR on input",
        IGNCR ,     "Ignore CR",
        ICRNL ,     "Map CR to NL on input",
        IXON ,      "Enable start/stop output control",
        /* _IXANY ,   "enable any char to restart output", */
        IXOFF ,       "Enable start/stop input control",
        0 ,           NULL };

struct flaginfo local_flags[] = {
        ISIG ,      "Enable signals",
        ICANON ,    "Canonical input (erase and kill)",
        /* _XCASE ,   "Canonical upper/lower appearance", */
        ECHO ,      "Enable echo",
        ECHOE ,     "Echo ERASE as BS - SPACE - BS",
        ECHOK ,     "Echo KILL by starting new line",
        0 ,         NULL };

show_some_flags( struct termios *ttyp )
/*
 * show the values of two of the flag sets_: c_iflag and c_lflag
 * adding c_oflag and c_cflag is pretty routine - just add new
 * tables above and a bit more code below.
 */
{
   show_flagset( ttyp->c_iflag, input_flags );
   show_flagset( ttyp->c_lflag, local_flags );
}
show_flagset( int thevalue, struct flaginfo thebitnames[] )
/*
 * check each bit pattern and display descriptive title
 */
{
   int i;
   for ( i=0; thebitnames[i].fl_value ; i++ ) {
       printf( " %s is ", thebitnames[i].fl_name);
       if ( thevalue & thebitnames[i].fl_value )
          printf("ON\n");
       else
          printf("OFF\n");
   }
}
```

showtty 用来显示驱动程序里 16 个属性的当前状态，并附有注释。程序使用了结构表以简化代码。一个简单的函数 show_flagset 接收一个整数和驱动程序标志集。如在这个程序中增加其他的标志集需要些什么呢？将这个程序转换成完整版本的 stty 需要些什么呢？

5.5.8 终端连接小结

终端是人们用来和 Unix 进程进行通信的设备。终端拥有一个可以让进程读取字符的键盘和可让进程发送字符的显示器。终端是一个设备，所以它在目录树中表现为一个特殊的文件，通常在/dev 这个目录中。

进程和终端间的数据传输和数据处理由终端驱动程序负责，终端驱动程序是内核的一部分。该内核代码提供缓冲、编辑和数据转换。程序可通过调用 tcgetattr 和 tcsetattr 查看和修改该驱动程序的设置。

5.6 其他设备编程：ioctl

到磁盘文件的连接有一个属性集，到终端的连接有另外一个属性集。到其他类型设备的连接是怎样的呢？

考虑 CD 刻录机。可擦写的 CD 能够被删除内容，能以不同的速度刻录 CD。扫描仪有自己的设置，如解析度和颜色深度等。其他类型的设备有各自的属性集。程序员如何查看和控制一个设备的设置呢？

每个设备文件都支持系统调用 ioctl：

	ioctl	
目标	控制一个设备	
头文件	#include <sys/ioctl.h>	
函数原型	int result = ioctl (int fd, int operation [, arg...]);	
参数	fd	与设备相联的文件描述符
	operation	需进行的操作
	arg...	操作所需参数
返回值	−1	发生错误
	other	依设备而定

系统调用 ioctl 提供对连接到 fd 的设备驱动程序的属性和操作的访问。每种类型的设备都有自己的属性集和 ioctl 操作集。

例如，一个终端屏幕，有一个以行和列或者是以像素为单位的大小属性。下面的代码显示屏幕的尺寸。

```
#include  <sys/ioctl.h>
void print_screen_dimensions()
{
   struct winsize wbuf;
```

```
    if ( ioctl ( 0, TIOCGWINSZ, &wbuf) != -1){
        printf ("%d rows x %d cols\n", wbuf.ws_row, wbuf.ws_col);
        printf ("%d wide x %d tall\n", wbuf.ws_xpixel, wbuf.ws_ypixel);
    }
}
```

符号 TIOCGWINSZ 是函数代码，wbuf 的地址是该设备控制函数的参数。

阅读头文件是了解设备类型以及相关函数的好方法。联机帮助中有关设备的内容也包括属性和函数的列表。例如，Linux 中有关 st(4)的联机帮助讲述了使用 ioctl 控制 SCSI 磁带驱动器的细节。

5.7　文件、设备和流

任何数据的源或目的地都被 Unix 视为文件。基本的系统调用既适用于磁盘文件也同样适用于设备文件，它们的区别体现在对连接的操作上。磁盘文件的文件描述符包含对缓冲属性和扩展属性的定义代码。终端的文件描述符包含编辑、回显、字符转换和其他操作的属性定义代码。

可以把每个处理步骤描述成连接的一个属性，但是反过来说，连接也可以看作是处理步骤的组合。20 世纪 80 年代由 AT&T 开发的一个 Unix 版本 System V 建立了一个以处理序列为基础的数据流模型。这有点像清洗汽车。首先，将肥皂水洒在车上。然后用大刷子擦去灰尘。下一步，用高压软管将表面的肥皂泡沫和灰尘清除，同时使用车盘锈抑制剂，打上热蜡，为轮轴盖镀铬。最后用软布和热空气弄干表面。

当然，每个步骤都是汽车清洗者从汽车清洗公司买来部件完成的。买来之后把它们安置在清洗序列中的某个位置，整个系统就可以工作了。而且，可以进行重组和改造，比如省略某些特定的步骤(注意请不要用热蜡!)。

这是数据流模型和连接属性的流模型的一个大概的想法。流模型的一个重要特征是处理的模块化。如果不满意仅能支持像大小写转换这样的终端驱动程序，可以设计并安装一个可将数字转换成罗马数字的模块。也就是，可以编写一个能完成从阿拉伯数字到罗马数字转换的处理模块。将它写到流模型规范，然后使用特殊的系统调用将该模块安装到系统上。流数据经过它的处理就从阿拉伯数字变成罗马数字了。

在联机帮助上查看 streamio 以了解更多关于通过这种方式管理连接属性问题的解决方案。在某些版本的 Unix 中，流用来实现网络服务。

小　　结

1. 主要内容

- 内核在进程和外部世界间交换数据。外部世界包括磁盘文件、终端和外部设备(像打印机、磁带驱动器、声卡和鼠标)。到磁盘文件和终端的连接有相似之处但也有差异。
- 磁盘文件和设备文件都有名字、属性和权限位。标准文件系统调用 open、read、

write、close 和 lseek 可被用于任何文件或设备。文件权限位以同样的方式应用于控制设备文件和磁盘文件的访问。

- 到磁盘文件的连接在处理和传输数据方面不同于到设备文件的连接。内核中管理与设备连接的代码被称为设备驱动程序。通过使用 fcntl 和 ioctl，进程可以读取和改变设备驱动程序的设置。
- 到终端的连接是如此的重要，以至函数 tcgetattr 和 tcsetattr 专门用来提供对终端驱动器的控制。
- Unix 命令 stty 使得用户能够访问 tcgetattr 和 tcsetattr 函数。

2. 图示

进程使用 write 将数据写入文件描述符，用 read 从文件描述符读出数据。文件描述符可被连接到磁盘文件、终端和外部设备。文件描述符指向设备驱动程序时，设备驱动程序具有属性设置，如图 5.13 所示。

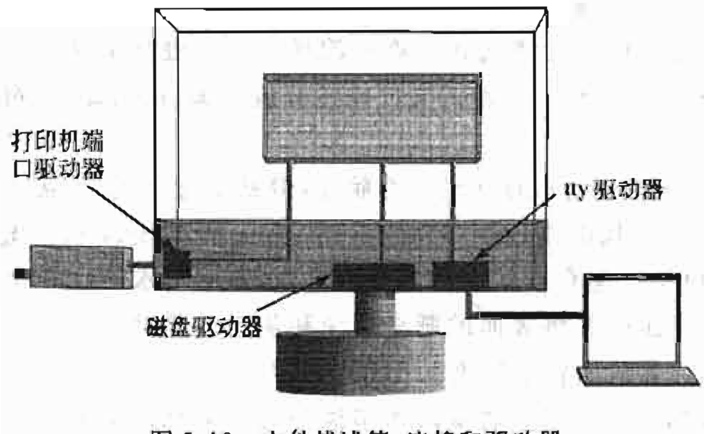


图 5.13　文件描述符、连接和驱动器

3. 下一章的内容

从磁盘读取数据相对容易，但是从用户终端读取有点麻烦，因为人是不可预知的。需要用户输入数据的程序可以利用终端驱动器的一些特别的连接控制功能。在下一章中将详细了解一些有关用户程序编写方面的主题。

4. 习题

5.1　在一个 Linux 机器上，很容易读取鼠标的输出。做这个工作需要处于文本模式。在 shell 中，确保称为 gpm 的程序不在运行：输入 gpm -k。然后，输入 cat /dev/mouse。然后移动鼠标并按键。命令 cat 从设备文件中读取数据。从该文件读取的字节是鼠标产生的按键次数和移动消息。

5.2　设备文件中的执行位是什么意思？学习命令 biff，考虑这个位的作用。

5.3　前面已经讨论了设备文件的输入/输出如何运作。那么像 ln、mv 和 rm 等的目录操作如何运作呢？利用图 5.1，解释这三个命令是如何影响目录、i-节点和驱动程序的。

5.4　命令 rm 和系统调用 unlink 删除与 i-节点的一个链接。如果该 i-节点的链接数降为 0，则内核释放磁盘块和 i-节点。设备 i-节点没有分配列表和数据块。取而代之的是设备文件的 i-节点包含指向内核设备驱动子程序的指针。如果删除设备的文件名，则内核释放 i-节点，驱动程序仍旧在内核中。如何新创建一个文件连接到该设备呢？（提示：阅读 mknod）

5.5　考虑为文件添加内容时的竞争情况。前文中的讨论描述了一个可能的顺序。这两个进程每个进程有两个操作，那么有多少种顺序组合的可能性呢？每种顺序的结果是什么？

5.6　查看 Linux 内核代码，找出 O_APPEND 位是在何处被校验的。自动定位是如何实现的？

5.7　系统调用 rename 是个原子操作。在这个单一的调用中合并了哪些步骤呢？查看 Unix 的内核代码，以了解所有的竞争情况和内核处理的可能冲突。Linux 代码中的解释有些饶舌和有趣。

5.8　标准库函数 fopen 支持以添加模式打开文件。例如，fopen ("data", "a")。在你的系统上，这是通过添加模式开启 O_APPEND，还是仅仅在打开文件后定位文件的末尾来实现的呢？找到 fopen 的源代码，或者编写一个程序添加模式两次打开同一个文件，然后交替向两个流写数据。根据所发生的现象能得到什么结论？

5.9　例程 echostate.c 报告文件描述符 0 表示的驱动器回显位的状态。使用重定向符“<”将标准输入定向到其他的文件或设备。尝试以下试验：

```
echostate < /dev/tty
echostate < /dev/lp
echostate < /etc/passwd
echostate < 'tty'
```

说出每个命令行的输出。

5.10　程序 setecho 改变附加在标准输入的驱动程序的回显位。如果将标准输入重定向到一个不同的终端，就可以改变该终端的回显位。

尝试以下试验。

(1) 在同一台机器上登录 2 次（或在机器上打开 2 个窗口）。

(2) 在每个窗口中输入 tty，以找到那两个窗口的设备文件名。假设一个连接到/dev/ttyp1，另一个连接到/dev/ttyp2。

(3) 在 ttyp1，输入 **setecho n < /dev/ttyp2**。

(4) 在 ttyp2 窗口，输入命令 **echostate**。

(5) 然后在 ttyp1，输入 **echostate < /dev/ttyp2**。

(6) 解释发生的情况。

(7) 用命令 stty 做同样的试验。

可能有一天，你会发现这个功能的确很有用。

5.11　前文中的例子为 tcgetattr 和 tcsetattr 调用使用的文件描述符的值为 0。第一个参数是文件描述符，也可以是任何指向到终端设备连接的值。
文件描述符 1 指向标准输出。修改 echostate 和 setecho，使用文件描述符 1 代替 0。这个变化如何影响程序的执行？通常标准输入和标准输出指向终端。解释 echostate >> echostate.log 将发生什么？使用文件描述符 0 有什么好处呢？

5.12　如果想建立一个接收 ppp 连接的系统，需要安装一个调制解调器，并且配置串行端口。串行接口的终端驱动程序应该被配置成能和调制解调器协同工作。阅读文件/etc/gettydefs 和/etc/inittab，学习 Unix 如何为在串行线上的登录定义终端设置。

5.13　有些 Unix 支持三种版本的 O_SYNC：仅数据块、仅 i-节点和两者都是。为什么需要确保以某种方式写入？控制这 3 种类型的标志的名各是什么？

5.14　在一个终端特殊文件上的读和写的权限位用来控制什么？使用 tty 确定你的终端的名字，然后使用 chmod 000 /dev/yourtty 将你的终端设置成对你来说也不可读。此时发生了什么情况？为什么呢？

5.15　查看你系统上的/dev 目录，找到不支持 read 操作的文件，不支持 write 操作的文件以及不支持 lseek 操作的文件。

5.16　在/dev 中使用 ls -l 找到各种设备的最大数目和最小数目。你看到了什么模式？什么设备具有同样的最大数目？这些设备有什么共同点？它们如何区分？

5.17　为 tty 驱动器设置的 4 个组命名。解释每个组的目的，并在每个组中命名两个位。

5.18　程序使用 tcsetattr 关闭当前终端的回显模式。当那个程序存在，终端处于无回显状态。另一方面，当程序打开一个文件并使用 fcntl 将描述符置于 O_APPEND 模式，下一个打开这个文件的程序不能获得自动添加模式。解释这个明显的不一致的地方。

5.19　到终端的连接是个正规的文件描述符。你能够使用 fcntl 为与文件描述符的连接设置 O_APPEND 属性吗？自动添加对设备来说是什么意思？

5.20　ioctl 和 fcntl 间的区别是什么？

5.21　目录/dev 包含文件/dev/null 和/dev/zero。这些文件并不是到设备的连接，但是它们也并不代表磁盘文件。这些文件是用来做什么的？为什么它们是有用的？你能够在/dev 目录中找到虚拟设备的其他文件吗，就像这两个文件一样？

5. 编程练习

5.22　改进这章中 write 的简单版本。前文中的版本要求用户输入设备文件名，而且该版本并不显示不同的欢迎用语。编写一个新版本，能够接受用户名作为参数，并在屏幕上显示你想通信的用户。查看 write 的标准版本显示的内容。

你的程序应该可以处理这样的特殊情况：你想聊天的人可能没有登录。另一方面，你想聊天的人可能登录了若干个终端。

5.23　不想被那些发送 write 的人打扰的用户可以使用命令 mesg。阅读 mesg，通过程序试验以了解它如何工作，然后写一个这样的程序。

5.24　通常的竞争情况包括两个进程同时更新同一个文件。例如，当你在一些系统上修改你的密码，程序 passwd 重写文件/etc/passwd。如果两个用户同时修改他们的密码会怎样呢？

一种防止对一个文件进行同时访问的方法是利用系统调用 link 的一个重要性质。考虑以下代码：

```
/*
 * tries to make a link called /etc/passwd.LCK
 * returns 0 if ok, 1 if already locked, 2 if other problem
 */
int lock_passwd()
{
    int rv = 0;                     /* default return value */
    if ( link ("/etc/passwd", "/etc/passwd.LCK") ==  -1)
        rv = ( errno == EEXISTS ? 1 :2 );
    return rv;
}
```

(1) 如果两个进程在同一时刻执行这段代码，只有一个会成功。系统调用 link 如何使用有效的方法给文件上锁？

(2) 使用这种方法编写一个程序，将文本的一行添加到一个文件。你的程序需要尝试建立链接。如果链接成功，程序能够打开文件，添加行，然后删除链接。如果建立链接失败，你的程序需要使用 sleep(1)等待 1 s，然后再次尝试。你需要确保你的程序不会永远处于等待状态。

(3) 写一个不用 lock_passwd 的 unlock_passwd 函数。

(4) 本例显示了允许进程给一个已存在的文件上锁，但是如何使用 link 编程避免两个进程创建同样的文件呢？

(5) 学习命令 vipw。vipw 为锁使用链接吗？

5.25　前一个问题显示了如何使用链接给文件上锁。当给文件上锁的程序结束修改文件时，文件的锁必须被删除。如果程序不释放锁，其他的程序将会永远处于等待状态。如果程序有漏洞，在它释放锁之前崩溃，或被用户按下 Ctrl-C 结束，将会

怎样？

一种解决方法是拥有锁的程序每隔 n 秒修改文件。程序可以使用 utime 去做。等待锁的程序可查看修改时间，检查锁是否依然有效。如果锁在上述规定的间隔中未被修改，那么其他的程序可随意删除链接，然后再次创建链接。

编写一个 lock_passwd 的新版本，使之带有表示秒数的数字参数。这个新版本能够实现上面所描述的逻辑。

5.26 如果关闭缓冲，将会有什么影响？编写一个程序，用来将一个大的磁盘文件分装在小的片里面，例如 2MB 的文件被分装在 16 字节的片的集合中。将 O_SYNC 开启和关闭进行试验。改变文件和分片的大小，以查看它们如何影响结果。

5.27 前文包含了关闭文件描述符磁盘缓冲的代码。编写一个将缓冲重新开启的程序。

5.28 编写一个称为 uppercase.c 的程序，用来跟踪终端驱动器里的 OLCUC 位，并报告该位的当前状态。

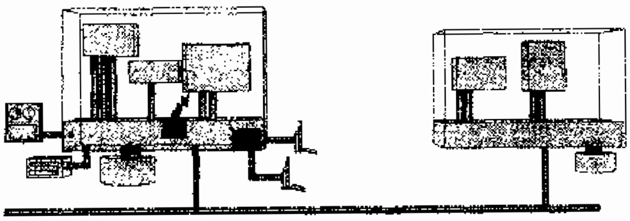
5.29 stty -a 的输出包含终端窗口的行数和列数。这些值不是来自 tcgetattr，而是来自 ioctl。使用这个系统调用修改第 1 章的 more，使它能够使用终端窗口的大小显示，而不是固定值 24。

6. 项目

基于本章的内容，能够了解和编写以下这些 Unix 程序：

write、stty、passwd、wall、biff、mt(磁带控制程序，可能你的系统上没有)

第 6 章　为用户编程：终端控制和信号



概念与技巧

- 软件工具与用户程序
- 读取和修改终端驱动程序的设置
- 终端驱动程序的模式
- 非阻塞输入
- 用户输入的超时
- 对信号的介绍：Ctrl + C 是如何工作的

相关的系统调用

- fcntl
- signal

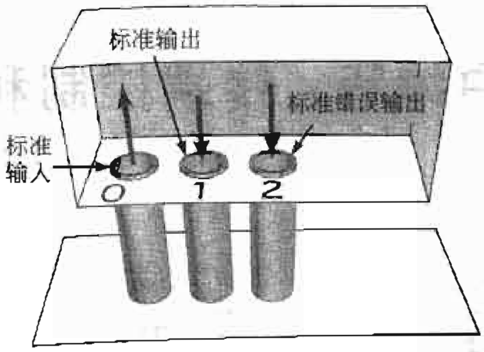
6.1　软件工具与针对特定设备编写的程序

在 Unix 系统，虽然设备看起来很像磁盘文件，但是设备是不同于磁盘文件的。在第 5 章中已经看到，程序能对设备执行 open、close、read、write 和 lseek 操作，但是也已经看到设备有相应的驱动程序，那些驱动程序包含许多与设备相关的控制和属性。程序如何认识这种双重性？

1. 软件工具：从 stdin 或文件读入，写到 stdout

对磁盘文件和设备文件不加以区分的程序被称为软件工具。Unix 系统有好几百个软件工具，包括 who、ls、sort、uniq、grep、tr 和 du。软件工具使用图 6.1 所示的模型。

软件工具从标准输入读取字节，进行一些处理，然后将包含结果的字节流写到标准输出。工具发送错误消息到标准错误输出，它们也被当做简单的字节流来处理。这些文件描述符能够连接到文件、终端、鼠标、光电管、打印机和管乐器；工具对所处理的数据的源和目的地不做任何假设。其他很多程序也能从命令行所指定的文件中读取数据。



事实：大多数进程自动将前3个文件描述符打开。它们不需要调用 open() 来建立连接

图 6.1　3 种标准文件描述符

这些程序的输入和输出能够被重定向到任何类型的连接上：

```
$ sort > outputfile
$ sort x > /dev/lp
$ who | tr '[a-z]' '[A-Z]'
```

2. *特定设备程序：为特定应用控制设备*

其它程序（如控制扫描仪、记录压缩盘、操作磁带驱动程序和拍摄数码相片的程序）也能同特定设备进行交互。在本章中将通过了解最常见的与特定设备相关的程序（通过终端与人交互的程序）来探讨在写这些程序时用到的概念和技术。将这些面向终端的程序称为用户程序。

3. *用户程序：一种常见的设备相关程序*

用户程序的例子有 vi、emacs、pine、more、lynx、hangman、robots 和许多加利福尼亚大学伯克利分校编写的游戏程序[①]。这些程序设置终端驱动程序的击键和输出处理方式。驱动程序有很多设置，但是用户程序常用到的有：

(1) 立即响应击键事件
(2) 有限的输入集
(3) 输入的超时
(4) 屏蔽 Ctrl-C

下面将通过编写一个实现所有这些特点的程序来学习这些主题。

6.2　终端驱动程序的模式

首先讨论在前面章节中提到的终端驱动程序。下面通过一个简短的转换程序来深入理解设备驱动程序的细节[②]：

---

① 这些程序的源代码可以在网上找到，寻找 bsdgames。
② 可以用 tr 命令做同样的事情，但是 tr 的 GNU 版本有输入缓冲，这样用它来做教学的例子就不是很好了。

```
/* rotate.c : map a->b, b->c, .. z->a
 *   purpose:useful for showing tty modes
 */

#include <stdio.h>
#include <ctype.h>
int main()
{
    int c;
    while ( ( c = getchar() ) != EOF ){
        if ( c == 'z' )
            c = 'a';
        else if (islower(c))
            c++;
        putchar(c);
    }
}
```

6.2.1 规范模式：缓冲和编辑

使用默认设置运行这个程序(<- 是退格键)：

```
$ cc rotate.c -o rotate
$ ./rotate
abx<-cd
bcde
efgCtrl-C
$
```

图 6.2 显示了终端、内核、rotate 程序和数据流。

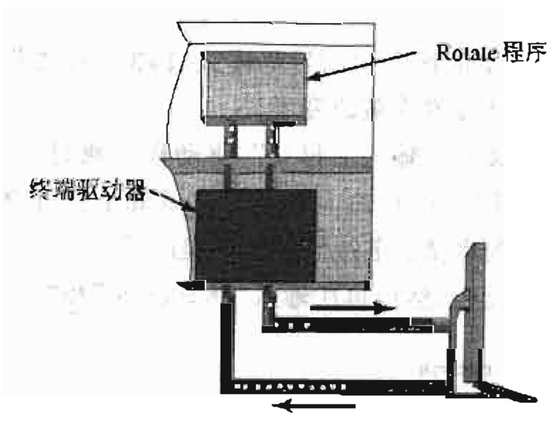


图 6.2 输入的内容和程序所得到的内容

上述的实验揭示了标准输入处理的如下特征：

(1) 程序未得到输入的"x",这是因为退格键删除了它;

(2) 击键的同时字符显示在屏幕上,但是直到按了回车键,程序才接收到输入;

(3) Ctrl-C 键结束输入并终止程序。

程序 rotate 不做这些操作。缓冲、回显、编辑和控制键处理都由驱动程序完成。图 6.3 显示了驱动程序中的操作层次。

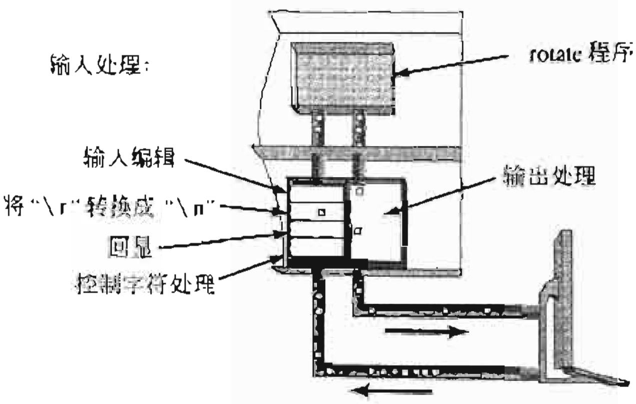


图 6.3　终端驱动器中的处理层

缓冲和编辑包含规范处理(canonical processing)。当这些特征被启动,终端连接被称为处于规范模式。

6.2.2　非规范处理

现在,尝试这个试验(输入仍旧是 abx<- cd,然后,输入 efg Ctrl-C):

```
$ stty - icanon; . /rotate
abbcxy^? cdde
effggh
$ stty icanon
```

命令 stty -icanon 关闭了驱动程序中的规范模式处理。上例并没有展示非规范模式的各个侧面,而只是演示了输入处理方式被改变了。

特别地,非规范模式没有缓冲。输入字母"a",驱动程序跳过缓冲层,将字符直接送到程序 rotate,然后程序显示字符"b"。用户输入未被缓冲可能是一件麻烦事。当用户试图删除一个字符,驱动程序不能做任何事情;字符早就送给程序了。

最后一个试验,尝试以下命令,然后再次输入"abx<- cd"和"efg"Ctrl-C:

```
$ stty - icanon - echo ; . /rotate
bcy^? de
fgh
$ stty icanon echo (注意:你看不到这个。为什么?)
```

在这个例子中关闭了规范模式和回显模式。驱动程序不再显示所输入的字符。输出

仅来自程序。当退出这个程序时，驱动程序仍旧处于无回显、非规范模式中，并且一直处于那种状态直到程序改变了设置。shell 打印一个显示符，等待下一行命令。有些 shell 重置驱动程序，而有些不这么做。如果 shell 并不重置驱动程序，将继续处于无回显、非规范的模式中。

6.2.3 终端模式小结

如果还未在终端尝试这些例子，现在就做。这些例子演示了终端驱动程序的不同模式。当为 Unix 设计用户程序时，需要决定哪种终端模式适合这个应用。

1. 规范模式

规范模式，也被称为 cooked 模式，是用户常见的模式。驱动程序输入的字符保存在缓冲区，并且仅在接收到回车键[①]时才将这些缓冲的字符发送到程序。缓冲数据使驱动程序可以实现最基本的编辑功能，如删除字符、单词或整行。当用户分别按下删除键、单词删除键或是终止键时，这些功能就会被调用。被指派到这些功能的特定键在驱动程序里设置，可通过命令 stty 或系统调用 tcsetattr 来修改。

2. 非规范模式

当缓冲和编辑功能被关闭时，连接被称为处于非规范模式。终端处理器仍旧进行特定的字符处理，例如，处理 Ctrl-C 及换行符和回车符之间的转换。但是，用于删除、单词删除和终止的编辑键没有特殊的意义，因此相应的输入被视作常规的数据输入。

如果用非规范模式编写程序，并且希望用户能够编辑他们的输入，需要在你的程序中实现编辑功能。

3. raw 模式

每个处理步骤都被一个独立的位控制。例如，ISIG 位控制 Ctrl-C 键是否用于终止一个程序。程序可随意关闭所有这些处理步骤。

当所有处理都被关闭后，驱动程序将输入直接传递给程序。在这种情况下，驱动程序被称为处于 raw 模式。在终端驱动程序更为简单的老版本系统中，有个特定的模式被称为 raw 模式。命令 stty 支持 raw 模式，将它作为命令行的一个选项。联机帮助上有关 stty 的部分解释了 raw 模式的含义。

终端驱动程序是内核中一些复杂的程序。通过前面的学习和试验可以更为清楚地了解它们的各个组成部分和功能。图 6.4 显示了其主要部分。

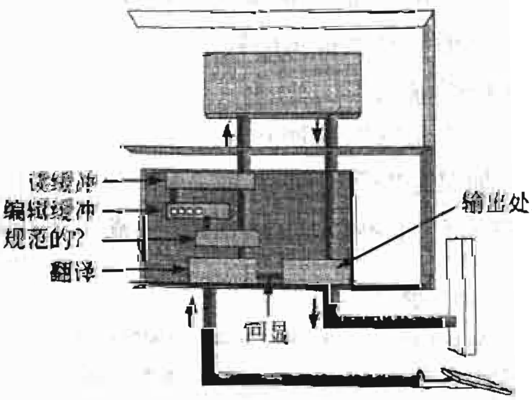


图 6.4 终端驱动程序的主要组成部分

① 或者是当前定义的 EOF 键，通常为 Ctrl-D。

各种模式都有各自特定的用途。为了帮助理解这些模式的实际应用价值，下面开发了一个使用不同模式的用户程序。

6.3　编写一个用户程序：play_again.c

很多用户应用程序，例如，自动取款机和计算机游戏，都会向用户提出 yes/no 的问题。以下程序脚本是一个银行应用程序的主循环：

```
#! /bin/sh
#
# atm.sh - a wrapper for two programs
#
while true
do
    do_a transaction        # run a program
    if play_again           # run our program
    then
        continue            # if"y" loop back
    fi
    break                   # if "n" break
done
```

就像其他典型的 Unix 风格的程序，这个银行脚本程序将程序的各个组件组合起来。第一个组件是一个称为 do_a_transaction 的程序，完成 ATM 的工作。第二个组件 play_again，从用户那儿得到"是"或"否"的回答。下面实现第二个组件程序。这样的组件架构允许方便地更换不同版本的 play_again。

play_again.c 的逻辑很简单：

- 对用户显示提示问题
- 接受输入
- 如果是"y"，返回 0
- 如果是"n"，返回 1

1. 例子：play_again0.c——完成上述功能

```
/* play_again0.c
 * purpose: ask if user wants another transaction
 * method: ask a question, wait for yes/no answer
 * returns: 0 =>yes, 1 =>no
 * better: eliminate need to press return
 */
#include  <stdio.h>
#include  <termios.h>

#define   QUESTION    "Do you want another transaction"
```

```
int get_response( char * );

int main()
{
    int response;
    response = get_response(QUESTION); /* get some answer */
    return response;
}
int get_response(char * question)
/*
 * purpose: ask a question and wait for a y/n answer
 * method: use getchar and ignore non y/n answers
 * returns: 0 =>yes, 1 =>no
 */
{
    printf("%s (y/n)?", question);
    while(1){
        switch( getchar() ){
            case 'y':
            case 'Y': return 0;
            case 'n':
            case 'N':
            case EOF: return 1;
        }
    }
}
```

这个程序显示提示问题，然后循环读取用户的输入，直到用户输入了“y”、“n”、“Y”或“N”才停止。play_again0 有两个问题，这两个问题都由运行时处在规范模式引起。首先，用户必须按回车键，play_again0 才能接受到数据。第二，当用户按回车键时，程序接收整行的数据并对其进行处理。因此，play_again0 把下面的输入作为一个否定的回答。

```
$ play_again0
Do you want another transaction (y/n) ? sure thing!
```

第一个改进是关闭规范输入，使得程序能够在用户敲键的同时得到输入的字符。

2. 例子：play_again1.c ——即时响应

```
/* play_again1.c
 * purpose: ask if user wants another transaction
 * method: set tty into char - by - char mode, read char, return result
 * returns: 0 =>yes, 1 =>no
 * better: do no echo inappropriate input
 */
#include  <stdio.h>
```

```
#include  <termios.h>

#define  QUESTION   "Do you want another transaction"

main()
{
    int  response;
    tty_mode(0);                              /* save tty mode */
    set_crmode();                             /* set chr-by-chr mode */
    response = get_response(QUESTION);        /* get some answer */
    tty_mode(1);                              /* restore tty mode */
    return response;
}
int get_response(char *question)
/*
 * purpose: ask a question and wait for a y/n answer
 * method: use getchar and complain about non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
    int input;
    printf("%s (y/n)?", question);
    while(1){
        switch( input = getchar() ){
            case 'y':
            case 'Y': return 0;
            case 'n':
            case 'N':
            case EOF: return 1;
            default:
                printf("\ncannot understand %c, ", input);
                printf("Please type y or no \n");
        }
    }
}
set_crmode()
/*
 * purpose: put file descriptor 0 (i.e. stdin) into chr-by-chr mode
 * method: use bits in termios
 */
{
    struct   termios   ttystate;
    tcgetattr( 0, &ttystate);                 /* read curr. setting */
    ttystate.c_lflag       &= ~ICANON;        /* no buffering */
    ttystate.c_cc[VMIN]     = 1;              /* get 1 char at a time */
```

```
    tcsetattr( 0 , TCSANOW, &ttystate);      /* install settings */
}
/* how == 0 => save current mode, how == 1 => restore mode */
tty_mode(int how)
{
    static struct termios original_mode;
    if ( how == 0 )
        tcgetattr(0, &original_mode);
    else
        return tcsetattr(0, TCSANOW, &original_mode);
}
```

play_again1 首先将终端置于一个字符接着一个字符的模式(character - by - character mode)，然后调用函数显示一个提示符，并获得一个响应，最后设置终端为原始的模式。注意，最后并未将终端置于规范模式。取而代之的是，将原先的设置复制到一个称为 original_mode 的结构中，结束时恢复这些设置。

将终端置于字符输入模式包括两部分工作。除了将 ICANON 位关闭外，还要将值 1 分配给控制字符数组中以 VMIN 为下标的元素。VMIN 的值告诉驱动程序一次可以读取多少个字符。因为希望一个接一个地读取字符，所以将这个值置为 1。如果希望一次读取 3 个字符[①]，则可将值置为 3。

编译并运行这个程序，输入 sure 作为回答：

```
$ make play_again1
cc play_again1.c  -o play_again1
$ ./play_again1
Do you want another transaction (y/n)? s
cannot understand s, Please type y or no
u
cannot understand u, Please type y or no
r
cannot understand r, Please type y or no
e
cannot understand e, Please type y or no
y$
```

像预期的那样，play_again1 接收和处理字符，而不再等待回车键。但是对每个非法字符都提示错误信息可能让人觉得比较烦。一个更好的设计是关闭回显模式，丢掉不需要的字符，直到得到可接受的字符为止。

---

① 实际上我用这个处理功能键。在很多键盘上，功能键发送多个字符序列，例如 escape 表示 - [- 1 - 1 ~。当键盘读取 escape 字符(ASCII 27)，它期待一次读入一行上的 3 个或 4 个字符。

3. 例子：play_again2.c——忽略非法键

```
/* play_again2.c
 * purpose: ask if user wants another transaction
 *  method: set tty into char-by-char mode and no-echo mode
 *          read char, return result
 * returns: 0=>yes, 1=>no
 *  better: timeout if user walks away
 *
 */
#include   <stdio.h>
#include   <termios.h>

#define  QUESTION "Do you want another transaction"

main()
{
   int response;
   tty_mode(0);                           /* save mode */
   set_cr_noecho_mode();                  /* set -icanon, -echo */
   response = get_response(QUESTION);     /* get some answer */
   tty_mode(1);                           /* restore tty state */
   return response;
}
int get_response(char *question)
/*
 * purpose: ask a question and wait for a y/n answer
 * method: use getchar and ignore non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
   printf("%s (y/n)?", question);
   while(1){
      switch( getchar() ){
         case 'y':
         case 'Y': return 0;
         case 'n':
         case 'N':
         case EOF: return 1;
      }
   }
}
set_cr_noecho_mode()
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
```

```
 * method: use bits in termios
 */
{
    struct termios    ttystate;
    tcgetattr( 0, &ttystate);                   /* read curr. setting */
    ttystate.c_lflag   &=  ~ICANON;             /* no buffering */
    ttystate.c_lflag   &=  ~ECHO;               /* no echo either */
    ttystate.c_cc[VMIN]     = 1;                /* get 1 char at a time */
    tcsetattr( 0 , TCSANOW, &ttystate);         /* install settings */
}
/* how == 0 => save current mode, how == 1 => restore mode */
tty_mode(int how)
{
    static struct termios original_mode;
    if ( how == 0 )
        tcgetattr(0, &original_mode);
    else
        return tcsetattr(0, TCSANOW, &original_mode);
}
```

这个程序和前面的版本在两个方面有所不同。设置终端驱动程序的函数关闭了回显位。注意，恢复函数不需要特意将该位开启。另一个变化是函数 get_response 不再提示错误信息，而仅仅是忽略它们。

编译并运行这个程序。如果输入 sure，没有显示任何内容。仅当输入 y 或 n 时，程序返回。

play_again2 如所希望的那样运行，但是它还有待改进。如果这个程序运行在真正的 ATM 上，而顾客在输入 y 或 n 之前走开了，将会怎样？下一个顾客跑过来按下 y，就能进入那个离开的顾客的账号。所以如果用户程序包含超时特征，会变得更安全。

非阻塞输入：play_again3.c

下一个程序版本含有超时特征。通过设置终端驱动程序，使之不等待输入来实现这个特征。先检查看是否有输入，如果发现没有输入，则先睡眠几秒钟，然后继续检查输入。如此尝试 3 次之后放弃。

1. 阻塞与非阻塞输入

当调用 getchar 或 read 从文件描述符读取输入时，这些调用通常会等待输入。在 play_again 例子中，对 getchar 的调用使得程序一直等待用户的输入，直到用户输入一个字符。程序被阻塞，直到能获得某些字符或是检测到了文件的末尾。那么如何关闭输入阻塞呢？

阻塞不仅仅是终端连接的属性，而是任何一个打开的文件的属性。程序可以使用 fcntl 或 open 为文件描述符启动非阻塞输入(nonblock input)。play_again3 使用 fcntl 为文件描

述符开启 O_NDELAY[①] 标志。

关闭一个文件描述符的阻塞状态并调用 read。结果如何呢？如果能够获得输入，read 获得输入并返回所获得的字符个数。如果没有输入字符，read 返回 0，这就像遇到文件末尾一样。如果有错误，read 返回－1。

非阻塞操作的内部实现相当简单。每个文件都有一块保存未读取数据的地方，如图 6.4 所示的驱动程序里最顶端的存储方块。如果文件描述符置了 O_NDELAY 位，并且那块空间是空的，read 调用返回 0。如果能阅读与 O_NDELAY 有关的的 Linux 源代码，就可以了解到实现的细节。

2. 例子：play_again3.c——使用非阻塞模式实现超时响应

```
/* play_again3.c
 * purpose: ask if user wants another transaction
 *  method: set tty into chr - by - chr, no - echo mode
 *          set tty into no - delay mode
 *          read char, return result
 * returns: 0 =>yes, 1 =>no, 2 =>timeout
 *  better: reset terminal mode on Interrupt
 */
#include  <stdio.h>
#include  <termios.h>
#include  <fcntl.h>
#include  <string.h>

#define  ASK  "Do you want another transaction"
#define  TRIES  3                          /* max tries */
#define  SLEEPTIME  2                      /* time per try */
#define  BEEP  putchar('\a')               /* alert user */

main()
{
    int response;
    tty_mode(0);                            /* save current mode */
    set_cr_noecho_mode();                   /* set  -icanon, -echo */
    set_nodelay_mode();                     /* noinput  => EOF */
    response = get_response(ASK, TRIES);    /* get some answer */
    tty_mode(1);                            /* restore orig mode */
    return response;
}
get_response( char *question , int maxtries)
/*
 * purpose: ask a question and wait for a y/n answer or maxtries
```

① 也可以使用 O_NONBLOCK 位，请查阅联机帮助。

```
 * method: use getchar and complain about non-y/n input
 * returns: 0=>yes, 1=>no, 2=>timeout
 */
{
    int input;
    printf("%s (y/n)?", question);          /* ask */
    fflush(stdout);                          /* force output */
    while ( 1 ){
        sleep(SLEEPTIME);                    /* wait a bit */
        input = tolower(get_ok_char());      /* get next chr */
        if ( input == 'y' )
            return 0;
        if ( input == 'n' )
            return 1;
        if ( maxtries-- == 0 )               /* outatime? */
            return 2;                        /* sayso */
        BEEP;
    }
}
/*
 * skip over non-legal chars and return y,Y,n,N or EOF
 */
get_ok_char()
{
    int c;
    while( ( c = getchar() ) != EOF && strchr("yYnN",c) == NULL )
        ;
    return c;
}
set_cr_noecho_mode()
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
 * method: use bits in termios
 */
{
    struct termios    ttystate;
    tcgetattr( 0, &ttystate);               /* read curr. setting */
    ttystate.c_lflag   &= ~ICANON;          /* no buffering */
    ttystate.c_lflag   &= ~ECHO;            /* no echo either */
    ttystate.c_cc[VMIN}    = 1;             /* get 1 char at a time */
    tcsetattr( 0 , TCSANOW, &ttystate);     /* install settings */
}
set_nodelay_mode()
/*
```

```
 * purpose: put file descriptor 0 into no-delay mode
 * method: use fcntl to set bits
 * notes: tcsetattr() will do something similar, but it is complicated
 */
{
    int termflags;
    termflags = fcntl(0, F_GETFL);          /* read curr. settings */
    termflags |= O_NDELAY;                  /* flip on nodelay bit */
    fcntl(0, F_SETFL, termflags);           /* and install 'em */
}
/* how == 0 => save current mode, how == 1 => restore mode */
/* this version handles termios and fcntl flags */
tty_mode(int how)
{
    static struct termios original_mode;
    static int original_flags;
    if ( how == 0 ){
        tcgetattr(0, &original_mode);
        original_flags = fcntl(0, F_GETFL);
    }
    else {
        tcsetattr(0, TCSANOW, &original_mode);
        fcntl( 0, F_SETFL, original_flags);
    }
}
```

程序的这个版本有一些新特点。首先使用 fcntl 关闭和开启非阻塞模式，其次在 get_response 中使用 sleep 和计数器 maxtries。

3. play_again3 的小问题

play_again3 并不是理想的。运行在非阻塞模式，程序在调用 getchar 给用户输入字符之前睡眠 2s。就算用户在 1s 内完成输入，程序也要在 2s 后才得到字符。用户可能会感到迷惑："我不是输入了吗？怎么没反映呢？难道说我弄错了？"

可以使程序更快地做出响应吗？可以减少每次调用 getchar 间的睡眠时间，并相应地增加循环次数来实现相同的超时设置。

另外，注意在显示提示符之后对 fflush 的调用。如果没有那一行，在调用 getchar 之前，提示符将不能显示。究其原因是，终端驱动程序不仅一行行地缓冲输入，而且还一行行地缓冲输出。驱动程序缓冲输出，直到它收到一个换行符或者程序试图从终端读取输入。在这个例子中，为了给用户读提示符的时间，需要延迟读入。这样就必须调用 fflush。

4. 实现超时的其他方法

Unix 提供更好的方法来实现超时功能，在驱动程序中设置数组 c_cc[]中的元素 VTIME 将超时功能的实现移至终端驱动程序。本章的一个练习提供了相关细节。系统调用 select 包含一个超时参数。将在后面的章节中讨论 select。

5. play_again3 的一个大问题

play_again3 忽略它所不想要的字母，识别和处理合法输入，并在规定的时间间隔内无合法输入的情况下自动退出。但是如果用户输入 Ctrl−C 将会如何？下面是它的运行情况：

```
$ make play_again3
cc play_again3.c  - o play_again3
$ . /play_again3
Do you want another transaction (y/n)? press Ctrl - C now
$ logout
Connection to host closed.
bash$
```

当按下 Ctrl−C 终止程序 play_again，不但终止了这个程序，同时也终止了整个登录会话。这是为什么呢？play_again3 有 3 个步骤：初始化、获得用户输入和恢复设置，如在图 6.5 中描述的那样。

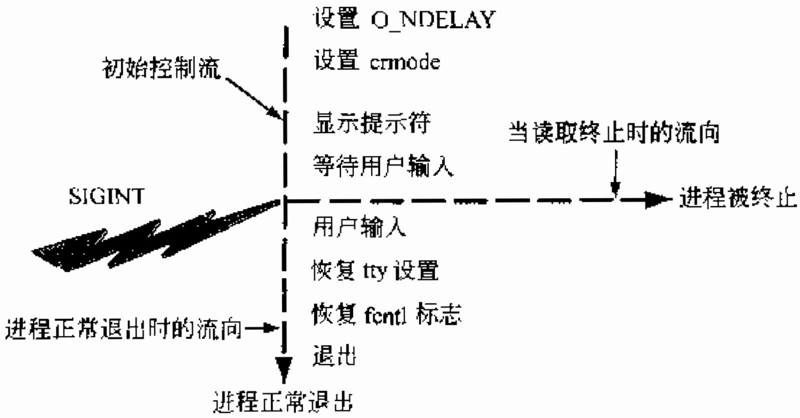


图 6.5　Ctrl − C 终止进程并将终端置于未恢复状态

初始化部分将终端置于非阻塞输入状态。程序随后进入主循环并打印提示，睡眠和读取输入。然后在程序运行中通过按 Ctrl−C 键来终止程序。那么终端驱动程序处于什么状态？

实际上程序会立刻退出，而不执行重置驱动程序的代码。当返回 shell 显示提示符并从用户处获得命令行时，终端仍旧处于非阻塞模式。shell 调用 read 获取命令行，但是因为处于非阻塞状态，read 立即返回 0。总之，程序结束时文件描述符处于一个错误的状态。下一个目标是学习如何避免程序被 Ctrl−C 鲁莽地杀死。

6. 为什么有些情况下不会被注销？

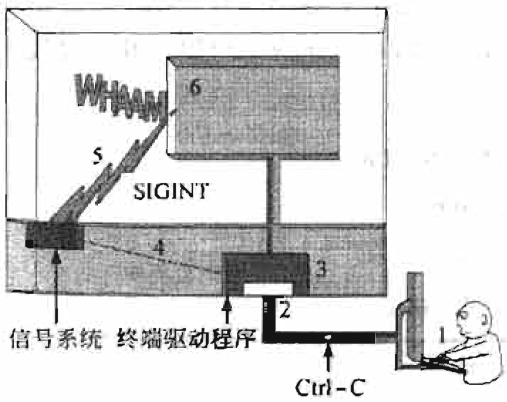
很多 Unix shell 包含编辑特征，如使用箭头键在执行过的命令列表中滚动。这些 shell 运行在实现这些功能所需要的 raw 模式下。当程序退出或死亡时，像 bash 和 tcsh 这些 shell 立即重置终端的属性。

6.4 信　　号

Ctrl-C 中断当前运行的程序。这个中断由一个称为信号的内核机制产生。信号是一个简单而重要的概念。下面将探讨信号的基本概念，学习怎样使用它们解决 play_again3 的问题。在下一章中将更深入地学习信号。

6.4.1 Ctrl－C 做什么

输入 Ctrl-C，程序便被终止了。一个单一的击键是如何杀死一个进程的呢？终端驱动程序在这里起了相应的作用，图 6.6 显示了相应的事件链。



1. 用户输入 Ctrl－C
2. 驱动程序收到字符
3. 匹配 VINTR 和 ISIG 的字符被开启
4. 驱动程序调用信号系统
5. 信号系统发送 SIGINT 到进程
6. 进程收到 SIGINT
7. 进程消亡

图 6.6　Ctrl－C 如何工作

中断信号的击键组合不一定非是 Ctrl－C，可以使用 stty（或者 tcsetattr）将当前的 VINTR 控制字符替换成另一种键。

6.4.2 信号是什么

按下 Ctrl-C 产生一个信号，那么信号是什么呢？信号是由单个词组成的消息。绿灯是一个信号，停止标牌是一个信号，裁判手势也是一个信号。这些物体和事件不是消息，走、停和出界才是消息。当按 Ctrl-C 时，内核向当前正在运行的进程发送中断信号。每个信号都有一个数字编码。中断信号通常是编码 2。[①]

信号从哪里来？信号来自内核，生成信号的请求来自 3 个地方，如图 6.7 所示。

(1) 用户

用户能够通过输入 Ctrl-C、Ctrl-\，或是终端驱动程序分配给信号控制字符的其他任何键来请求内核产生信号。

(2) 内核

当进程执行出错时，内核给进程发送一个信号，例如，非法段存取、浮点数溢出，或是一

① 若改变它，许多 shell 脚本将出问题。

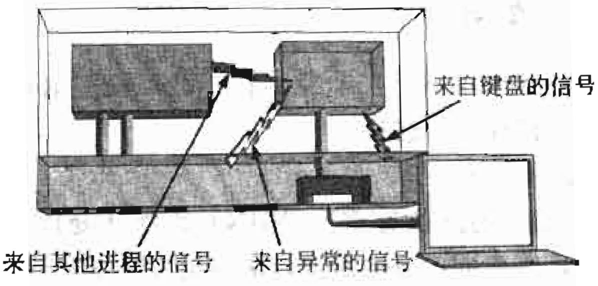


图 6.7　信号的 3 种来源

个非法的机器指令。内核也利用信号通知进程特定事件的发生。

(3) 进程

一个进程可以通过系统调用 kill 给另一个进程发送信号。一个进程可以和另一个进程通过信号通信。

由进程的某个操作产生的信号被称为同步信号(synchronous signals)，例如，被零除。由像用户击键这样的进程外的事件引起的信号被称为异步信号(asynchronous signals)。

哪里可以找到信号的列表？信号编号以及它们的名字通常出现在/usr/include/signal.h 文件中。这里是这个文件的一部分：

```
#define SIGHUP   1    /* hangup,generated when terminal disconnects */
#define SIGINT   2    /* interrupt,generated from terminal special char */
#define SIGQUIT  3    /* ( * ) quit, generated from terminal special char */
#define SIGILL   4    /* ( * ) illegal instruction (not reset when caught) */
#define SIGTRAP  5    /* ( * ) trace trap (not reset when caught) */
#define SIGABRT  6    /* ( * ) abort process */
#define SIGEMT   7    /* ( * ) EMT instruction */
#define SIGFPE   8    /* ( * ) floating point exception */
#define SIGKILL  9    /* kill (cannot be caught or ignored) */
#define SIGBUS   10   /* ( * ) bus error (specification exception) */
#define SIGSEGV  11   /* ( * ) segmentation violation */
#define SIGSYS   12   /* ( * ) bad argument to system call */
#define SIGPIPE  13   /* write on a pipe with no one to read it */
#define SIGALRM  14   /* alarm clock timeout */
#define SIGTERM  15   /* software termination signal */
```

例如，中断信号被称为 SIGINT，退出信号被称为 SIGQUIT，非法段存取信号是 SIGSEGV。任何一个版本的 Unix 的手册都包含更多的相关信息。在 Linux 中，可以查看 signal(7)的相关联机帮助。

信号做什么？这要视情况而定。很多信号杀死进程。某时刻进程还在运行，下一秒它就消亡了，从内存中被删除，相应的所有的文件描述符被关闭，并且从进程表中被删除。使用 SIGINT 消灭一个进程，但是进程也有办法保护自己不被杀死。

6.4.3 进程该如何处理信号

当进程接收到 SIGINT 时，并不一定非要消亡。进程能够通过系统调用 signal 告诉内核，它要如何处理信号。进程有 3 个选择。

(1) 接受默认处理(通常是消亡)

手册上列出了对每个信号的默认处理。SIGINT 的默认处理是消亡。进程并不一定要使用 signal 接受默认处理，但是进程能够通过以下调用来恢复默认处理：

```
signal (SIGINT, SIG_DFL);
```

(2) 忽略信号

程序可以通过以下调用来告诉内核，它需要忽略 SIGINT 信号：

```
signal (SIGINT, SIG_IGN);
```

(3) 调用一个函数

第 3 种选择是 3 个当中最强大的一种。考虑 play_again3 这个例子。当用户输入 Ctrl-C，当前运行的程序立即退出，而不调用恢复驱动程序设置的函数。更好的做法是，程序在接收到 SIGINT 后，调用一个恢复设置的函数，然后再退出。

调用 signal 的第 3 种选择允许这种类型的响应。程序能够告诉内核，当信号到来时应该调用哪个函数。在信号到来时被调用的函数被称为信号处理函数。为安装信号处理函数，程序调用：

```
signal (signum, functionname);
```

	signal	
目标	简单的信号处理	
头文件	#include <signal.h>	
函数原型	result = signal (int signum, void (* action)(int));	
参数	signum	需响应的信号
	action	如何响应
返回值	-1	遇到错误
	prevaction	成功返还

调用 signal 为编码是 signum 的信号安装新的信号处理函数。action 可以是函数名或以下两个特殊值之一。

- SIG_IGN，忽略信号；
- SIG_DFL，将信号恢复为默认处理。

signal 返回前一个处理函数。值是指向函数的指针。

6.4.4 信号处理的例子

1. 捕捉信号

```
/* sigdemo1.c - shows how a signal handler works.
 * - run this and press Ctrl-C a few times
 */
#include  <stdio.h>
#include  <signal.h>

main()
{
    void f(int);                    /* declare the handler */
    int i;
    signal( SIGINT, f );            /* install the handler */
    for (i = 0; i<5;i++){           /* do something else */
        printf("hello\n");
        sleep(1);
    }
}
void f(int signum)                  /* this function is called */
{
    printf("OUCH! \n");
}
```

主函数由两部分组成，调用 signal 后进入一个循环。sigdemo1.c 调用 signal 来设置 SIGINT 的处理函数 f。如果进程接收到 SIGINT 信号，内核会调用函数 f 来处理这个信号。程序跳转到那个函数，执行它的代码，然后返回到跳转前的位置，就像子过程调用一样。

图 6.8 显示了两个独立的控制流：一个是正常的路径，进入 main，执行循环，然后从 main 返回；另一个是由信号引起的路径，进入 f，然后返回。

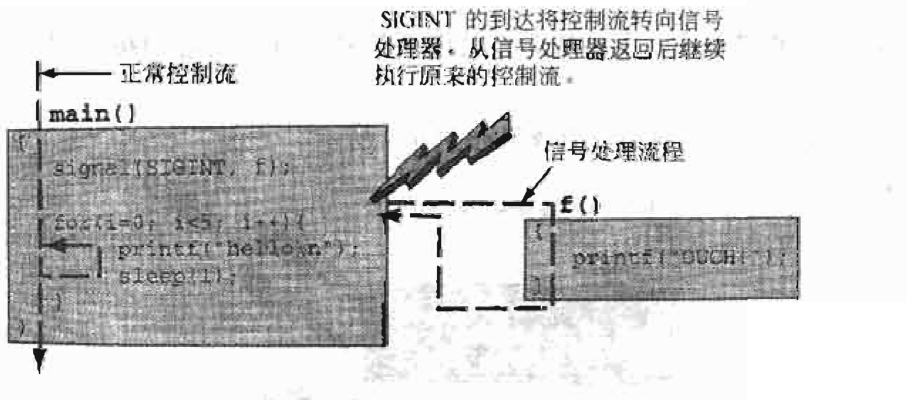


图 6.8　信号引起子过程的调用

以下是程序运行情况：

```
$ ./sigdemo1
hello
hello        press Ctrl - C now
OUCH!
hello        press Ctrl - C now
OUCH!
hello
hello
$
```

试编译并运行这个程序。程序中没有对 f 的显式调用。接受到的信号引发了对那个函数的调用。

2. 忽略信号

```
/* sigdemo2.c  - shows how to ignore a signal
 *             - press Ctrl - \ to kill this one
 */
#include   <stdio.h>
#include   <signal.h>

main()
{
    signal( SIGINT, SIG_IGN );
    printf("you can't stop me! \n");
    while( 1 )
    {
        sleep(1);
        printf("haha\n");
    }
}
```

sigdemo2.c 调用 signal 来设置为忽略中断信号。可以随意的按 Ctrl－C 而不会对进程产生影响。

signal(SIGINT，SIG_IGN)的效果如图 6.9 所示。

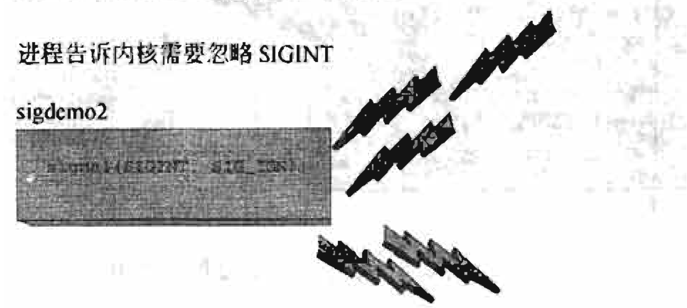


图 6.9　signal (SIGINT，SIG_IGN)的效果

以下是程序的运行情况：

```
$ ./sigdemo2
you can't stop me!
haha
haha
haha      press Ctrl - C now
haha      press Ctrl - C nowpress Ctrl - C now
haha
haha
haha      press ^\now
Quit
$
```

按键组合 Ctrl－\会发送一个不同的信号，即 quit 信号，这个程序没有忽略或捕捉 SIGQUIT。

6.5　为处理信号做准备：play_again4.c

现在已经知道如何修改 play_again3.c 来处理信号，但是还需要做一个设计决策。需要忽略信号并让用户回答 yes 或 no 吗？要捕捉键盘信号吗？当用户输入 no 时，要直接退出还是先返回一个值表示程序已经消亡？

下面这个程序版本捕捉 SIGINT，重置驱动程序，然后返回 no 的代码。

```
/* play_again4.c
 * purpose: ask if user wants another transaction
 *  method: set tty into chr - by - chr, no - echo mode
 *      set tty into no - delay mode
 *      read char, return result
 *      resets terminal modes on SIGINT, ignores SIGQUIT
 * returns: 0 => yes, 1 => no, 2 => timeout
 *  better: reset terminal mode on Interrupt
 */
#include        <stdio.h>
#include        <termios.h>
#include        <fcntl.h>
#include        <string.h>
#include        <signal.h>

#define   ASK         "Do you want another transaction"
#define   TRIES       3                   /* max tries */
#define   SLEEPTIME   2                   /* time per try */
#define   BEEP        putchar('\a')       /* alert user */
```

```
main()
{
    int response;
    void ctrl_c_handler(int);

    tty_mode(0);                              /* save current mode */
    set_cr_noecho_mode();                     /* set -icanon, -echo */
    set_nodelay_mode();                       /* noinput => EOF */
    signal( SIGINT, ctrl_c_handler );         /* handle INT */
    signal( SIGQUIT, SIG_IGN );               /* ignore QUIT signals */
    response = get_response(ASK, TRIES);      /* get some answer */
    tty_mode(1);                              /* reset orig mode */
    return response;
}
get_response( char *question , int maxtries)
/*
 * purpose: ask a question and wait for a y/n answer or timeout
 * method: use getchar and complain about non-y/n input
 * returns: 0=>yes, 1=>no
 */
{
    int input;

    printf("%s (y/n)?", question);             /* ask */
    fflush(stdout);                            /* force output */
    while ( 1 ){
        sleep(SLEEPTIME);                      /* wait a bit */
        input = tolower(get_ok_char());        /* get next chr */
        if ( input == 'y' )
           return 0;
        if ( input == 'n' )
           return 1;
        if ( maxtries-- == 0 )                 /* outatime? */
           return 2;                           /* sayso */
        BEEP;
    }
}
/*
 * skip over non-legal chars and return y,Y,n,N or EOF
 */
get_ok_char()
{
    int c;
    while( ( c = getchar() ) != EOF && strchr("yYnN",c) == NULL )
        ;
```

```
    return c;
}
set_cr_noecho_mode()
/*
 * purpose: put file descriptor 0 into chr - by - chr mode and noecho mode
 * method: use bits in termios
 */
{
    struct termios ttystate;

    tcgetattr( 0, &ttystate);                /* read curr. setting */
    ttystate.c_lflag   &=  ~ICANON;          /* no buffering */
    ttystate.c_lflag   &=  ~ECHO;            /* no echo either */
    ttystate.c_cc[VMIN]   =  1;              /* get 1 char at a time */
    tcsetattr( 0 , TCSANOW, &ttystate);      /* install settings */
}
set_nodelay_mode()
/*
 * purpose: put file descriptor 0 into no - delay mode
 * method: use fcntl to set bits
 * notes: tcsetattr() will do something similar, but it is complicated
 */
{
    int termflags;
    termflags  =  fcntl(0, F_GETFL);         /* read curr. settings */
    termflags |=  O_NDELAY;                  /* flip on nodelay bit */
    fcntl(0, F_SETFL, termflags);            /* and install 'em     */
}
/* how == 0 => save current mode, how == 1 => restore mode */
/* this version handles termios and fcntl flags          */

tty_mode(int how)
{
    static struct termios original_mode;
    static int            original_flags;
    static int            stored  =  0;

    if ( how == 0 ){
       tcgetattr(0, &original_mode);
       original_flags  =  fcntl(0, F_GETFL);
       stored  =  1;
    }
    else if ( stored ) {
       tcsetattr(0, TCSANOW, &original_mode);
       fcntl( 0, F_SETFL, original_flags);
```

```
    }
}
void ctrl_c_handler(int signum)
/*
 * purpose: called if SIGINT is detected
 * action: reset tty and scram
 */
{
    tty_mode(1);
    exit(1);
}
```

其他的设计留作练习。

6.6 进 程 终 止

程序使用 signal 来告诉内核它需要忽略哪些信号。如果有人编写了一个将所有类型的信号设置为 SIG_IGN 的程序，然后执行一个无限循环将会如何呢？

幸好，对系统管理员（和程序员）来说，Unix 不可能让一个程序永不停止。有两个信号是不能被忽略和捕捉的。阅读手册或头文件中的信号列表，看看哪些信号是不可阻挡的。

6.7 为设备编程

现在已经了解了编写终端控制程序的三个方面。首先，学习了驱动程序的属性和如何控制连接。然后，学习了应用程序的特定需求，并调整驱动程序以满足这些需求。最后，学习了如何处理信号——中断的一种形式。

这三个方面对所有的设备都适用。考虑一块声卡或一个磁盘驱动程序。设备有许多种由设备驱动程序控制的设置，需要了解这些设置。同样，程序必须实现特定的功能，调整驱动程序以满足这些需求。最后，很多设备驱动程序会产生信号报告错误或特定事件。磁盘驱动程序可能在它结束从磁盘到内存数据块的复制时发送一个信号，程序必须能够对这些信号做出响应。

小　　结

1. 主要内容

- 有些程序处理从特定设备来的数据。这些与特定设备相关的程序必须控制与设备的连接。Unix 系统中最常见的设备是终端。
- 终端驱动程序有很多设置。各个设置的特定值决定了终端驱动程序的模式。为用户编写的程序通常需要设置终端驱动程序为特定的模式。
- 键盘输入分为 3 类，终端驱动程序对这些输入做不同的处理。大多数键代表常规数

据，它们从驱动程序传输到程序。有些键调用驱动程序中的编辑函数。如果按下删除键，驱动程序将前一个字符从它的行缓冲中删除，并将命令发送到终端屏幕，使之从显示器中删除字符。最后，有些键调用处理控制函数。Ctrl－C 键告诉驱动程序调用内核中某个函数，这个函数给进程发送一个信号。终端驱动程序支持若干种处理控制函数，它们都通过发送信号到进程来实现控制。

- 信号是从内核发送给进程的一种简短消息。信号可能来自用户、其他进程或内核本身。进程可以告诉内核，在它收到信号时需要做出怎样的响应。

2. 进一步的问题

Unix 系统总是从很多终端或其他设备接收数据。用户可能在任何时刻输入数据，内核必须处理这些输入。Unix 系统同时运行若干个程序。内核如何在同一时刻维护多个并发的任务并对多个不可预知的中断作出响应呢？下一章将通过编写一个计算机游戏程序来探讨这个问题。

3. 习题

6.1　很多 Unix 软件工具从命令行指定的文件中读取数据。命令 tr 不是这样。tr 是用来干什么的？能想出它不接受命令行指定文件名的原因吗？还有其他仅从标准输入读取数据而不从指定的文件中读取数据的 Unix 工具吗？大多数的 Unix 命令存储在/bin、/usr/bin 和/usr/local/bin 目录中。

6.2　任何文件描述符都有 O_NDELAY 这个属性，而不仅仅是终端驱动程序的属性。这意味着这个属性适用于磁盘文件，也适用于设备文件。
对磁盘文件来说，非阻塞性意味着什么？除了终端文件，非阻塞性对设备意味着什么？

4. 编程练习

6.3　文件描述符可能处于阻塞或无延迟（即非阻塞）模式。终端驱动程序提供了其他选择，它允许对输入设置超时间隔。驱动程序 termios 结构体中的控制字符数组 c_cc[]在位置 VTIME 的元素是用来设置以毫秒为单位的超时间隔。因此，s. c_cc[VTIME] = 20 会将驱动程序的超时设置为 2 s。
改写 play_again3. c，使得它使用驱动程序中的超时特征，而不再将文件描述符置于非阻塞模式。

6.4　在 play_again 中处理信号。
(1) 修改 play_again3. c，使得它忽略键盘信号，而仅对 yes 或 no 做出响应。
(2) 修改 play_again3. c，使得它在收到键盘信号后，重置终端属性并以返回值 2 退出。

6.5　修改 rotate. c，使得它能够改变 tty 的模式。修改过的程序应该关闭规范模式，并且关闭回显。然后它读取字符并显示字母表中的下一个字母。当用户按下字母“Q”时，程序应该恢复 tty 设置并退出。
你的程序应该忽略键盘信号或通过在退出之前重置驱动程序来对它们进行处理。

6.6 编写一个行编辑器。编写运行在非规范模式程序的一个问题是缺乏输入编辑。修改纠正过的 rotate.c,使之支持字符和行编辑。特别地,当程序收到一个退格或删除字符时,它从屏幕上删除前一个字符。为删除一个字符,程序需要打印一个退格字符、一个空格字符,然后又一个退格字符。

同时,修改程序,使得它能够像终端驱动程序那样处理行删除字符。也就是,从屏幕上删除当前输入行的所有字符。

需要做什么以实现驱动程序的单词删除功能呢?

6.7 修改程序 sigdemo1.c,使得它能够对用户按下的 Ctrl-C 个数进行计数。修改过的程序将显示 OUCH!,然后 OUCH!!,即感叹号的个数和处理函数被调用的次数相等。

除了显示递增的感叹号,程序应该能够接受一个整数作为命令行参数。在用户按下 Ctrl-C 的次数与那个整数相等之后,程序应该退出。

6.8 修改程序 sigdemo1.c,使得它能够询问用户是否真的要终止程序。运行的样例如下所示:

```
hello
hello
   Interrupted! OK to quit (y/n)? n
hello
hello
   Interrupted! OK to quit (y/n)? y
$
```

如果当程序在等待用户回答 OK to quit (y/n)? 问题时,用户按下 Ctrl-C 将会发生什么事?实现上述程序,并观察发生的现象。

6.9 程序能够使用 signal 告诉内核它需要忽略特定的信号,像 SIGINT 和 SIGQUIT。另一种不同的方法是一开始就杜绝这些信号的产生。终端驱动程序有一个称为 ISIG 的标志。阅读手册,以了解这个标志的用途。然后使用这个标志重写 sigdemo2.c。

当修改过的程序接收到从键盘以外的地方发送来的 SIGINT 信号时,程序将做什么?学习命令 kill,并使用 kill 发送 SIGINT 到关闭了 ISIG 的那个版本的程序。

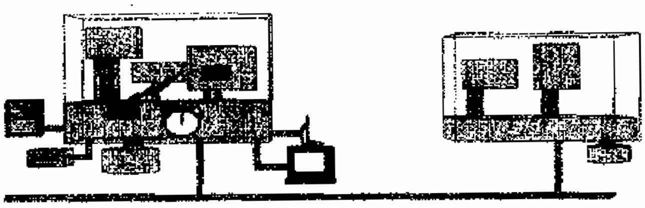
6.10 中断并不总是破坏性的。想象你正在做一个需要若干天的项目。你可能收到老板询问工作进展的电话。这些中断是用来调用状态报告的子程序的,而不是用来杀死进程。编写一个执行耗时任务的 C 程序。例如,编写一个使用较慢的方法寻找素数的程序。程序需要跟踪它到目前为止所找到的最大的素数。实现一个 SIGINT 的处理器函数,这个函数用来显示它所找到的素数的个数以及素数的最大值。

这个想法如何运用在系统编程中?

6.11　在第 1 章中，实现了几个版本的 more。那时并不知道如何控制终端驱动程序。改进那个程序，使它运行无回显、非规范模式中，并能对中断和终止信号做出正确的响应。

6.12　用户不仅能够通过按键产生信号，也能通过改变终端窗口的大小产生信号。窗口每次改变大小时，SIGWINCH 就被发送到进程。按默认处理，进程将忽略 SIGWINCH 信号。编写一个程序，使得能够在屏幕上打印满屏的字母“A”。例如，如果窗口有 10 行 20 列，程序要显示字母“A”200 次。当窗口大小改变时，程序用字母“B”填充整个屏幕。下一次，使用“C”，以此类推。当用户按下字母“Q”时，屏幕清除，程序退出。当用户按下其他任何键时，屏幕从字母“A”重新开始。

第 7 章　事件驱动编程：编写一个视频游戏



概念与技巧

- 异步事件驱动编程
- curses 库：目标和使用
- 警告和间隔计时器
- 可靠的信号处理
- 可重入代码、临界区
- 异步输入

相关的系统调用和函数

- alarm、setitimer、getitimer
- kill、pause
- sigaction、sigprocmask
- fcntl、aio_read

7.1　视频游戏和操作系统

贝尔实验室的 Dennise Ritchie 和 Ken Thompson 为了玩一个叫做《星际旅行》的视频游戏，于是编写了 Unix。Ritchie 写道：

Thompson 在 1969 年就开发了《星际旅行》这个游戏。第一次是在 Multics 操作系统上，然后用 Fortran 移植到 GECOS(GE 上运行的一个操作系统，后来在 Honeywell 635 上运行）。那是一个由玩家在模拟的移动太阳系中，根据屏幕上显现的环境引导飞船在不同的行星或卫星上降落的游戏。GECOS 上的这个版本在两个重要方面并不令人满意：第一，游戏的显示有些不连贯，同时通过敲击命令来控制很难掌握；第二，游戏需要在一台大型机上大约花费 75 美元的 CPU 时间来运行。所以不久以后，Thompson 找到一台很少使用的 PDP-7 计算机，这台机器有很棒的显示处理器，整个系统被当做 Graphic-II 的终端。他和我重写了《星际旅行》使之能运行在这台机器上。实际上，真正的目标比看上去更加雄心勃勃。因为我们抛弃了所有现存的软件，所以不得不写一个浮点运算包，显示特性的点序规格说明，以及一个在屏幕角上连续显示输入内容的排错子系统。所有这些都是用汇编语言写的，用一个在 GECOS 上的交叉汇编器编译到 PDP-7 的纸带上。

《星际旅行》尽管是一个非常吸引人的游戏，但也只是主要作为学习复杂的 PDP-7 程序开发的敲门砖。不久以后 Thompson 开始实现一个之前就已设计好的纸带文件系统（paper file system）（或者说粉笔文件系统即 chalk file system 更加合适）。在没有练习的情况下试图建立一个文件系统是非常困难的，所以他继续以一个实用操作系统的其他功能来充实自己的系统，尤其是进程的概念。然后是一小部分用户级的工具：复制、打印、删除和编辑文件的工具。当然还有一个简单的命令解释器（shell）。直到此时所有的程序还是在 GECOS 上写的，然后用纸带转移到 PDP-7 上，但是一旦有一个自己的汇编器它就可以支持自己了。尽管直到 1970 年 Brian Kernighan 才建议使用 Unix（某种程度上有点像 Multics 的双关语）来命名系统，如今所知这个操作系统已经诞生了。

视频游戏和操作系统有很多共同点。在本章中将完成一个简单的视频游戏。通过这个例子来介绍更多的 Unix 系统服务、一些基本原则和操作系统设计技术。

1. 视频游戏做什么

考虑一个有两个人参与的星际旅行视频游戏。程序创立行星、流星、飞船和其他物体的影像，并使它们移动。每一个物体有自己的移动速度、方向、动力和其他一些属性。物体之间相互作用。一个流星可能撞上飞船或其他流星。

游戏同时要响应用户输入。游戏玩家们通过按钮、鼠标和轨迹球在任何时刻都有可能生成输入，程序必须在很短的时间里作出响应。这些输入事件会影响游戏中物体的属性。通过按下按钮，用户可以增加飞船速度或是减少飞船质量。飞船的变化会影响它与其他物体的作用方式。

2. 视频游戏如何做

一个视频游戏综合了一些基本的概念和原则。

（1）空间

游戏必须在计算机屏幕的特定位置画影像。程序如何控制视频显示？

（2）时间

影像以不同的速度在屏幕上移动。以一个特定的时间间隔改变位置。程序是如何获知时间并且在特定的时间安排事情的发生？

（3）中断

程序在屏幕上平滑的移动物体，用户可在任意时刻产生输入。程序是如何响应中断的？

（4）同时做几件事

游戏必须在保持几个物体移动的同时还要响应中断。程序是如何同时做多件事情而不被弄得晕头转向的？

3. 操作系统面临类似的问题

操作系统同样要面对这 4 个问题。内核将程序载入内存空间并维护每个程序在内存中所处的位置。在内核的调度下，程序以时间片间隔的方式运行，同时，内核也在特定的时刻运行特定的内部任务。内核必须在很短的时间内响应用户和外设在任何时刻的输入。同时做几件事情需要一些技巧。内核是如何保证数据的有序和规整的？

4. 屏幕管理、时间、信号、共享资源

本章将学习屏幕管理、时间、信号和如何安全地同时做几件事情。为了学习这 4 个基本主题，将编写一个基于字符终端的动画游戏。

为什么是基于字符的图形界面？

为什么不用强大的 X11 来编程或者是 Java 来绘图？这里有很多原因。首先，基于字符的游戏除了每个像素大些外，其他与高分辨率的图形界面游戏非常相似。其次是可移植性。字符图形界面游戏只需要一个终端模拟器和一个连接，这些是每个系统都提供的。第三，在绘图上省下来的时间可以用在系统编程上。尽管如此，如果喜欢更好的图形界面，网站上有这个游戏的 X Windows 版本。

7.2　任务：单人弹球游戏(Pong)

现在开始了。本章的主要任务是实现一个在游戏房和家庭中常见的经典游戏——弹球游戏。图 7.1 显示了它的 3 个主要元素：墙、球和挡板。游戏的概要描述如下：

(1) 球以一定的速度移动；

(2) 球碰到墙壁或挡板会被弹回；

(3) 用户按按钮来控制挡板上下移动。

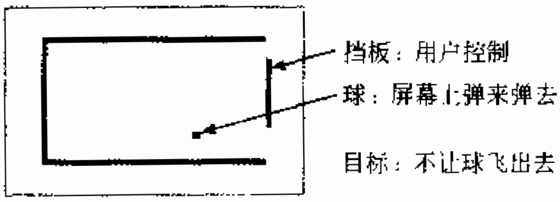


图 7.1　单人视频游戏

写这个游戏需要了解如何管理屏幕、时间和中断，还要理解如何安全地同时做几件事情。下面一样一样来学。

7.3　屏幕编程：curses 库

curses 库是一组函数，程序员可以用它们来设置光标的位置和终端屏幕上显示的字符样式。curses 库最初是由 UCB 的开发小组开发的。大部分控制终端屏幕的程序使用 curses。曾经由一组简单的函数组成的库现在包括了许多复杂的特性。这里将使用其中很少一部分的功能。

7.3.1　介绍 curses

curses 将终端屏幕看成是由字符单元组成的网格，每一个单元由(行、列)坐标对标示。坐标系的原点是屏幕的左上角，行坐标自上而下递增，列坐标自左向右递增。图 7.2 显示了 curses 屏幕。

curses 具有的函数包括可以将光标移动到屏幕上任何行、列单元，添加字符到屏幕或者从屏幕上删除字符，设置字符的可视属性(如颜色、亮度)，建立和控制窗口以及其他文本区域。用户手册有 curses 的所有函数的详细描述。这里将用到其中的 9 个。

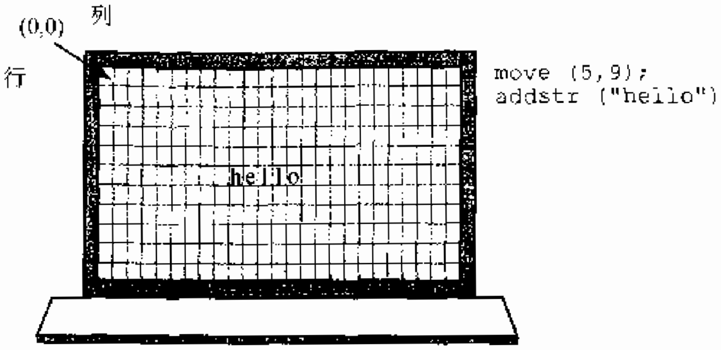


图 7.2　curese 视屏幕为网格

基本 curses 函数	
initscr()	初始化 curses 库和 tty
endwin()	关闭 curses 并重置 tty
refresh()	使屏幕按照你的意图显示
move(r,c)	移动光标到屏幕的(r,c)位置
addstr(s)	在当前位置画字符串 s
addch(c)	在当前位置画字符 c
clear()	清屏
standout()	启动 standout 模式(一般使屏幕反色)
standend()	关闭 standout 模式

curses 例 1：hello1.c

第一段程序展示了一个 curses 程序的基本逻辑：

```
/* hello1.c
 * purpose     show the minimal calls needed to use curses
 * outline     initialize, draw stuff, wait for input, quit
 */

#include  <stdio.h>
#include  <curses.h>

main()
{
    initscr() ;                          /* turn on curses*/
                                         /* send requests */
    clear();                             /* clear screen */
    move(10,20);                         /* row10,col20 */
    addstr("Hello, world");              /* add a string */
    move(LINES - 1,0);                   /* move to LL */
    refresh();                           /* update the screen */
    getch();                             /* wait for user input */
```

```
    endwin();                           /* turn off curses */
}
```

编译和运行程序非常简单：

```
$ cc hello1.c -l curses -o hello1
$ ./hello1
```

输出如图 7.3 所示。这个程序在连接到任何 Unix 系统的任何终端上都能运行。



图 7.3　第一个基于 curses 的程序

curses 例 2：hello2.c

将 curses 函数与循环、变量和其他函数组合在一起会产生更复杂的显示效果。预测以下第 2 个例子的输出：

```
/* hello2.c
 * purpose  show how to use curses functions with a loop
 * outline  initialize, draw stuff, wrap up
 */

#include  <stdio.h>
#include  <curses.h>

main()
{
    int i;

    initscr();                         /* turn on curses */
    clear();                           /* draw some stuff */
    for (i = 0; i<LINES; i++ ){        /* in a loop */
        move( i, i+i );
        if ( i%2 == 1 )
            standout();
        addstr("Hello, world");
        if ( i%2 == 1 )
            standend();
    }
```

```
    refresh();                    /* update the screen */
    getch();                      /* wait for user input*/
    endwin();                     /* reset the tty etc */
}
```

编译并运行它。你的预测正确吗?

7.3.2　curses 内部：虚拟和实际屏幕

refresh 函数做了些什么？试验一下：注释掉该行，重新编译，运行程序。结果是什么都没有出现在屏幕上。

curses 设计成为能够在不阻塞通信线路的情况下更新文本屏幕。curses 通过虚拟屏幕（如图 7.4 所示）来最小化数据流量。

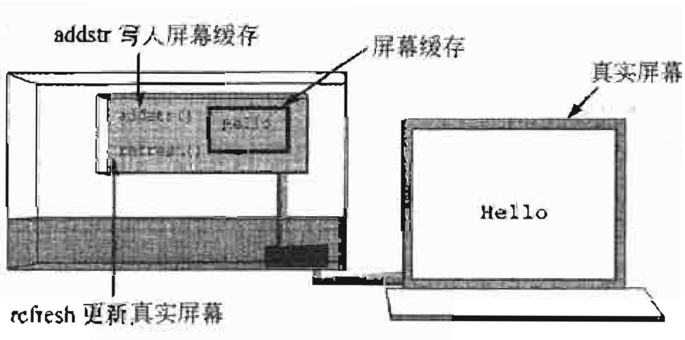


图 7.4　Curses 保持真实屏幕的副本

真实屏幕是眼前的一个字符数组。curses 保留了屏幕的两个内部版本。一个内部屏幕是真实屏幕的复制。另一个是工作屏幕，其上记录了对屏幕的改动。每个函数，比如 move、addstr 等都只在工作屏幕上进行修改。工作屏幕就像磁盘缓存，curses 中的大部分的函数都只对它进行修改。

refresh 函数比较工作屏幕和真实屏幕的差异。然后 refresh 通过终端驱动送出那些能使真实屏幕与工作屏幕一致的字符和控制码。例如，如果真实屏幕的左上角是 Smith、James，然后用 addstr 把 Smith、Jane 放在相同的位置，调用 refresh 也许只是用 n 和空格替换了 James 中的 m 和 s。这种只传输改变的内容而不是影像本身的技术被用在视频流中。

7.4　时钟编程：sleep

为了写一个视频游戏，需要把影像在特定的时间置于特定的位置。用 curses 把影像置于特定的位置。然后在程序中添加时间响应。第 1 步使用系统函数 sleep。

动画例子 1：hello3.c

```
/* hello3.c
 * purpose using refresh and sleep for animated effects
```

```
 * outline initialize, draw stuff, wrap up
 */
#include   <stdio.h>
#include   <curses.h>

main()
{
   int i;

   initscr();
     clear();
     for(i=0; i<LINES; i++ ){
         move( i, i+i );
         if ( i%2 == 1 )
                 standout();
         addstr("Hello, world");
         if ( i%2 == 1 )
                 standend();
         sleep(1);
         refresh();
     }
   endwin();
}
```

当编译并运行这个程序的时候，将看到 hello 字符串在屏幕自上而下逐行显示，每秒增加一行，反色和正常显示交替出现。为什么在每次循环结束都要调用 refresh? 如果不这样会有什么效果?

动画例子 2：hello4.c

```
/* hello4.c
 * purpose show how to use erase, time, and draw for animation
 */
#include   <stdio.h>
#include   <curses.h>

main()
{
   int  i;

   initscr();
     clear();
     for(i=0; i<LINES; i++ ){
         move( i, i+i );
         if ( i%2 == 1 )
                 standout();
         addstr("Hello, world");
         if ( i%2 == 1 )
```

```
                standend();
            refresh();
            sleep(1);
            move(i,i + i);                          /* move back */
            addstr("                   ");          /* erase line */
        }
    endwin();
}
```

hello4 创造移动的假象。字符串沿着对角线缓慢向下移动。秘诀是先在一个地方画字符串，睡眠 1 秒钟，然后在原来的地方画空字符串以删除原有影像，最后将输出位置推进。注意在两次请求之后通过调用 refresh 来保证每次循环后旧的影像消失，新的影像显示。图 7.5 是屏幕的一个快照。

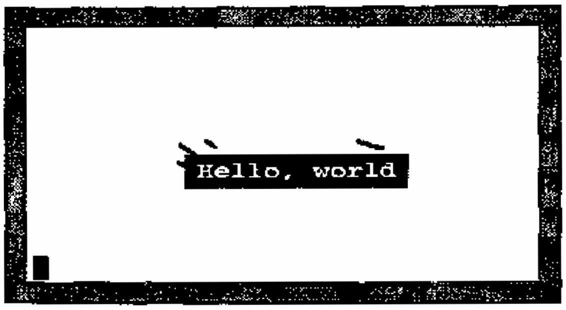


图 7.5　字符串慢慢向下移动

下一个例子将字符串在屏幕四壁弹来弹去。

动画例子 3：hello5.c

```
/* hello5.c
 * purpose bounce a message back and forth across the screen
 * compile cc hello5.c - lcurses - o hello5
 */
#include <curses.h>

#define LEFTEDGE    10
#define RIGHTEDGE   30
#define ROW         10

main()
{
char  message = "Hello";
char  blank   = "     ";
int   dir = +1;
int   pos = LEFTEDGE ;

initscr();
  clear();
```

```
    while(1){
      move(ROW,pos);
      addstr( message );              /* draw string */
      move(LINES - 1,COLS - 1);       /* park the cursor */
      refresh();                      /* show string */
      sleep(1);
      move(ROW,pos);                  /* erase string */
      addstr( blank );
      pos += dir;                     /* advance position */
      if ( pos >= RIGHTEDGE )         /* check for bounce */
         dir = -1;
      if ( pos <= LEFTEDGE )
         dir = +1;
    }
}
```

变量 dir 用来控制字符串移动的速度。当 dir 是+1 时，字符串每一秒向右移动一列。当 dir 是－1 时，字符串每秒向左移动一列。改变 dir 的符号就改变了字符串移动的方向。图 7.6 是某一特定时刻屏幕的快照。

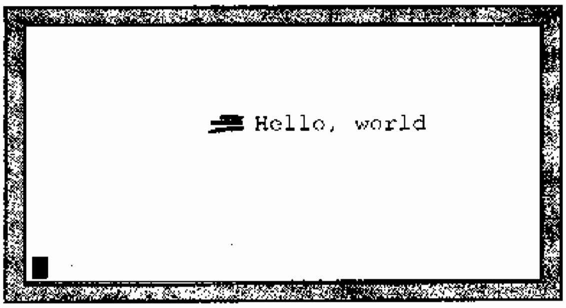


图 7.6　字符串弹来弹去

现在该如何做？

现在离能够写出一个像样的动作类视频游戏有多远？已经知道了如何在屏幕的任何地方画字符串，也知道了如何通过画、擦掉和重画之间插入时延来创造动画效果。迄今实现的程序还不错，只是：

(1) 一秒钟的时延太长，需要更精确的计时器；

(2) 需要增加用户输入。

这些问题引出新的话题：用时钟和高级信号编程。然后，将再次回到这个游戏。

7.5　时钟编程 1：Alarms

程序可以以不同的方式使用时钟。可以用来在执行流中加入时延。前面的 3 个例子里用 sleep 加入时延。时钟的另一个用途是调度一个将来要做的任务，就像拨好一个煮鸡蛋的定时器，然后干别的事情直到定时器鸣叫。同样的目的，Unix 提供 alarm。

7.5.1　添加时延：sleep

为了在程序中添加时延，使用 sleep 函数：

```
sleep(n)
```

sleep(n)将当前进程挂起 *n* 秒或者在此期间被一个不能忽略的信号的到达所唤醒。

7.5.2　sleep()是如何工作的：使用 Unix 中的 Alarms

sleep 函数的工作机理与你想睡定长时间的觉一样：

(1) 设置闹钟到你想睡的秒数；

(2) 睡觉，直到闹钟的铃声响起。

图 7.7 是这个机制的示意图。系统中的每个进程都有一个私有的闹钟(alarm clock)。这个闹钟很像一个计时器，可以设置在一定秒数后闹铃。时间一到，时钟就发送一个信号 SIGALRM 到进程。除非进程为 SIGALRM 设置了处理函数(handler)，否则信号将杀死这个进程。sleep 函数由 3 个步骤组成：

1. 为 SIGALRM 设置一个处理函数；
2. 调用 alarm(num_seconds)；
3. 调用 pause。

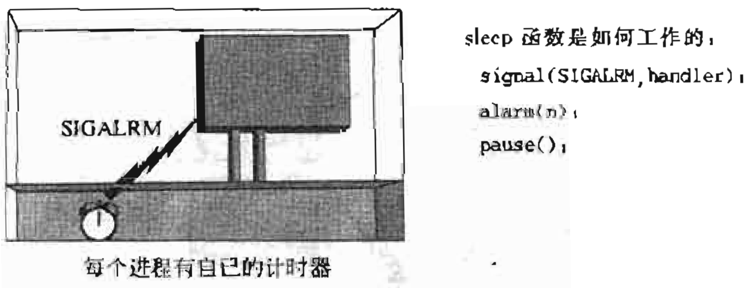


图 7.7　一个进程设置一个闹钟后挂起

系统调用 pause 挂起进程直到信号到达。任何信号都可以唤醒进程，而非仅仅等待 SIGALRM。以上想法总结为以下代码：

```
/* sleep1.c
 * purpose    show how sleep works
 * usage      sleep1
 * outline    sets handler, sets alarm, pauses, then returns
 */
#include  <stdio.h>
#include  <signal.h>
// #define SHHHH

main()
{
```

```
    void wakeup(int);

    printf("about to sleep for 4 seconds\n");
    signal(SIGALRM, wakeup);          /* catch it */
    alarm(4);                         /* set clock */
    pause();                          /* freeze here */
    printf("Morning so soon? \n");    /* back to work */
}

void wakeup(int signum)
{
#ifndef SHHHH
      printf("Alarm received from kernel\n");
#endif
}
```

这里调用 signal 设置 SIGALRM 处理函数，然后调用 alarm 设置一个 4 秒的计时器，最后调用 pause 等待。

调用 pause 的目的是挂起进程直到有一个信号被处理。当计时器计时 4 秒钟以后，内核送出 SIGALRM 给进程，导致控制从 pause 跳转到信号处理函数。在信号处理程序中的代码被执行，然后控制返回。当信号被处理完后，pause 返回，进程继续。图 7.8 总结了 pause 的执行过程。

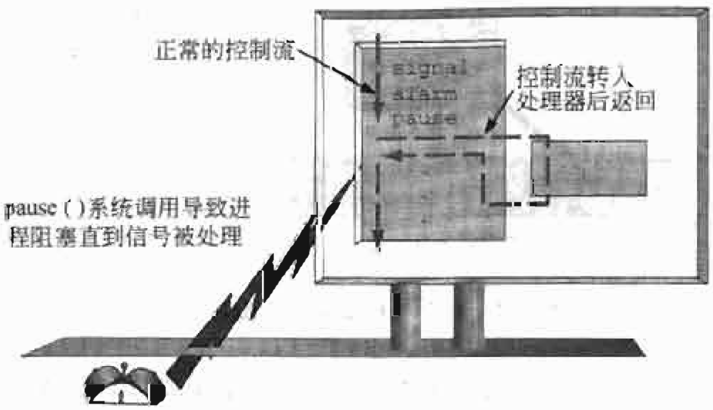


图 7.8　进入处理函数的执行流

下面是 alarm 和 pause 的细节：

	alarm	
目标	设置发送信号的计时器	
头文件	#include <unistd.h>	
函数原型	unsigned old = alarm(unsigned seconds)	
参数	seconds 等待的时间（秒）	
返回值	−1	如果出错
	old	计时器剩余时间

alarm 设置本进程的计时器到 seconds 秒后激发信号。当设定的时间过去之后，内核发送 SIGALRM 到这个进程。如果计时器已经被设置，alarm 返回剩余秒数(注意：调用 alarm(0)意味着关掉闹钟)。

pause	
目标	等待信号
头文件	#include <unistd. h>
函数原型	result = pause()
参数	没有参数
返回值	总是 −1

pause 挂起调用进程直到一个信号到达。如果调用进程被这个信号终止，pause 没有返回。如果调用进程用一个处理函数捕获，在控制从处理函数处返回后 pause 返回。这种情况下 errno 被设置为 EINTR。

7.5.3　调度将要发生的动作

计时器的另一个用途是调度一个在将来的某个时刻发生的动作同时做些其他事情。调度一个将要发生的动作很简单，通过调用 alarm 来设置计时器，然后继续做别的事情。当计时器计时到 0，信号发送，处理函数被调用。

7.6　时钟编程 2：间隔计时器

Unix 很早就有 sleep 和 alarm。它们所提供的时钟精度为秒，对于很多应用来说这个精度是不能让人满意的。后来一个更强大和使用广泛的计时器系统被添加进来。这个新的系统使用一个被称做为间隔计时器(interval timer)的概念，有更高的精度。而且每个进程都有 3 个独立的计时器而不是原来的一个。这还不是全部。每个计时器都有两个设置：初始间隔和重复间隔设置。新的系统还支持 alarm 和 sleep，它们对大多数应用来说已经足够了。图 7.9 是它的一个相应的示意图。

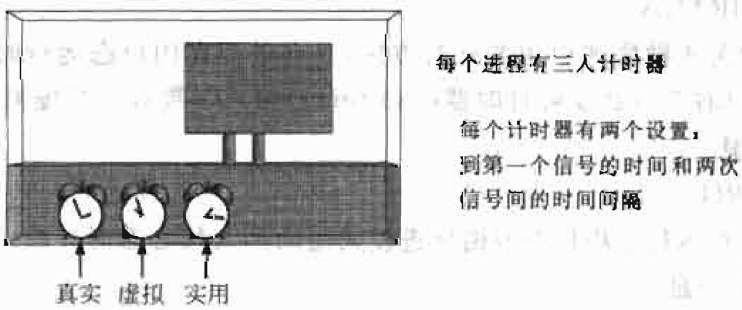


图 7.9　每个进程有 3 个计时器

可以用这个新的系统来添加时延和为事件定时。

7.6.1 添加精度更高的时延：usleep

为了添加精度更高的时延，使用 usleep：

```
usleep(n)
```

usleep(n)将当前进程挂起 n 微秒或者直到有一个不能被忽略的信号到达。

7.6.2 三种计时器：真实、进程和实用

进程可以以 3 种方式来计时。考虑一个程序在运行了 30 s 后结束。在一个分时系统中，这个程序不是一直在运行的，其他的程序与它共享处理器。图 7.10 显示了一种可能性。

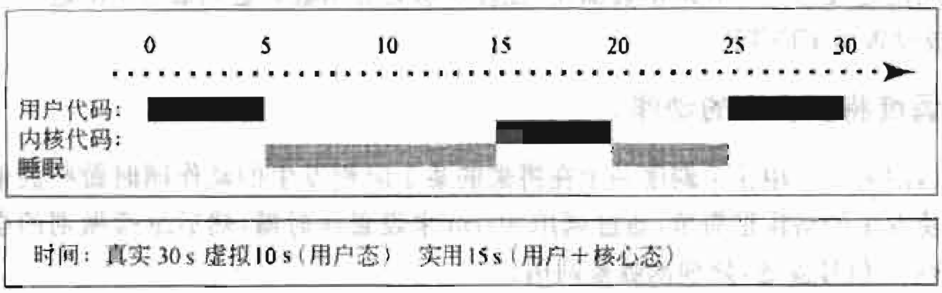


图 7.10 时间用在哪里

图 7.10 显示从 0 到 5 s 进程在用户模式运行，接着从 5 到 15 s 睡眠，然后在核心态运行到 20 s 睡眠，如此这般。显然从开始到结束，程序使用了 10 s 的用户时间、5 s 的系统时间。并显示了 3 种时间：真实时间、用户时间和用户时间＋系统时间。内核提供计时器来计量这 3 种类型的时间。3 类计时器的名字和功能如下：

(1) ITIMER_REAL

这个计时器计量真实时间，如同手表记录时间。也就是说不管程序在用户态还是核心态用了多少处理器时间它都记录。当这个计时器用尽，发送 SIGALRM 消息。

(2) ITIMER_VIRTUAL

这个计时器就像美式橄榄球中用的计时方法，只有进程在用户态运行时才计时。虚拟计时器(virtual timer)的 30 s 比实际计时器(real timer)的 30 s 要长。当虚拟计时器用尽，发送 SIGVTALRM 消息。

(3) ITIMER_PROF

这个计时器在进程运行于用户态或由该进程调用而陷入核心态时计时。当这个计时器用尽，发送 SIGPROF 消息。

7.6.3 两种间隔：初始和重复

医生给你一些药丸并告诉你："过一个小时吃第一粒，然后每隔 4 个小时吃一粒。"你需

要设置计时器到 1 个小时，然后在每次时尽后再设置为 4 个小时。每个间隔计时器的设置都有这样两个参数：初始时间和重复间隔。在间隔计时器用的结构体中初始时间是 it_value，重复间隔是 it_interval。如果不想要重复这一特征，将 it_interval 设置为 0。要把两个时钟都关掉，设 it_value 为 0。

7.6.4　用间隔计时器编程

程序中使用 alarm 不难，只要传给 alarm 秒数就可以了。程序中使用间隔计时器要复杂一点，要选择计时器的类型，然后需要选择初始间隔和重复间隔，还要设置在 struct itimerval 中的值。比如，为了使用间隔计时器来提醒你按 7.6.3 节规定的计划吃药，设置 it_value 为 1 小时，设置 it_interval 为 4 小时，然后将这个结构体通过调用 setitimer 传给计时器。为了读取计时器设置，使用 getitimer。图 7.11 是系统响应的示意图。

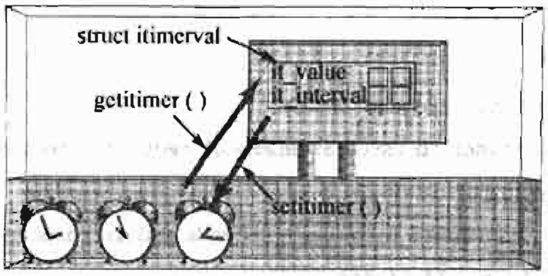


图 7.11　读写计时器设置

1. 间隔计时器例子：ticker_demo.c

程序 ticker_demo.c 演示了如何使用一个间隔计时器：

```
/* ticker_demo.c
 *    demonstrates use of interval timer to generate reqular
 *    signals, which are in turn caught and used to count down
 */

#include  <stdio.h>
#include  <sys/time.h>
#include  <signal.h>

int main()
{
   void countdown(int);

   signal(SIGALRM, countdown);
   if ( set_ticker(500) ==  -1 )
     perror("set_ticker");
   else
     while( 1 )
```

```
        pause();
    return 0;
}

void countdown(int signum)
{
    static int num = 10;
    printf("%d ..", num-- );
    fflush(stdout);
    if ( num < 0 ){
        printf("DONE! \n");
        exit(0);
    }
}

/* [from set_ticker.c]
 * set_ticker( number_of_milliseconds )
 * arranges for interval timer to issue SIGALRMs at regular intervals
 * returns -1 on error, 0 for ok
 * arg in milliseconds, converted into whole seconds and microseconds
 * note: set_ticker(0) turns off ticker
 */

int set_ticker( int n_msecs )
{
    struct itimerval new_timeset;
    long n_sec, n_usecs;

    n_sec = n_msecs / 1000 ;                        /* int part */
    n_usecs = ( n_msecs % 1000 ) * 1000L ;          /* remainder */

    new_timeset.it_interval.tv_sec = n_sec;         /*set reload */
    new_timeset.it_interval.tv_usec= n_usecs;       /*new ticker
                                                       value*/
    new_timeset.it_value.tv_sec    = n_sec;         /*store this*/
    new_timeset.it_value.tv_usec   = n_usecs;       /*and this*/

    return setitimer(ITIMER_REAL, &new_timeset, NULL);
}
```

来看看 ticker_demo.c 的程序流程。一开始使用 signal 设置函数 countdown 来处理 SIGALRM 信号。然后通过 set_ticker 来设置微秒数。

set_ticker 通过装载初始间隔和重复间隔设置间隔计时器。每个间隔是由两个值组成：秒数和微秒数，这就像实数的整数部分和小数部分。计时器开始计时，控制返回到 main。

回到 main，ticker_demo.c 进入一个无尽的循环，其间调用 pause。每过大约 500 μs，控制跳转到 countdown 函数。countdown 将一个静态变量的值递减，打印一条消息，通常情况下返回调用者。当变量 num 达到 0 时，countdown 调用 exit。

当然，main 不是一定要调用 pause 的。主程序可以做些其他更有趣的事情，这样在每个预定的时刻还是会跳转到 countdown 的。

间隔计时器的设置是通过 struct itimerval 来完成的。这个结构类型包括初始间隔和重复间隔，两者存储在 struct timeval 中：

```
struct itimerval
{
   struct timeval it_value;        /* time to next timer expiration*/
   struct timeval it_interval;     /* reload it_value with this */
}
struct timeval {
    time_t tv_sec                  /* seconds*/
    suseconds_t tv_usec;           /* and microseconds*/
}
```

不同的 Unix 版本，struct timeval 的细节可能有些差异。查一下你的系统的相应手册和头文件。

图 7.12 显示了结构中各成员的关系，图 7.13 显示了如何载入数据以使第一次计时器到达时间为 60.5 s，然后每 240.25 s 重复一次。

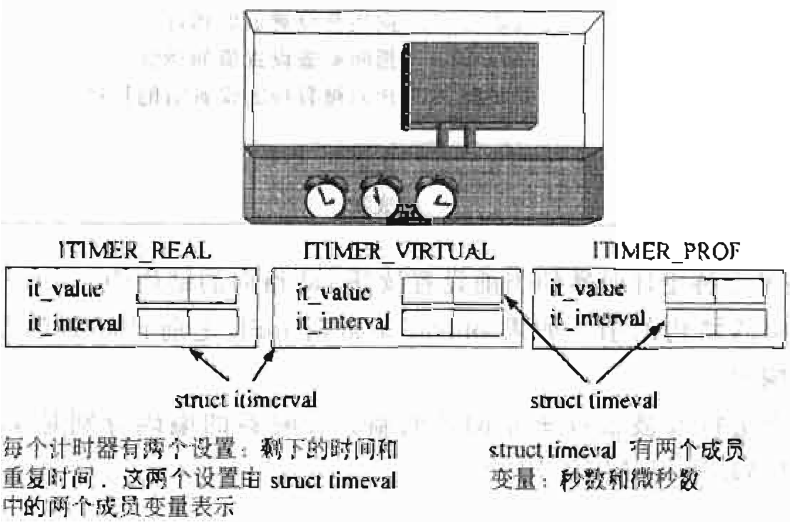


图 7.12　间隔计时器内部

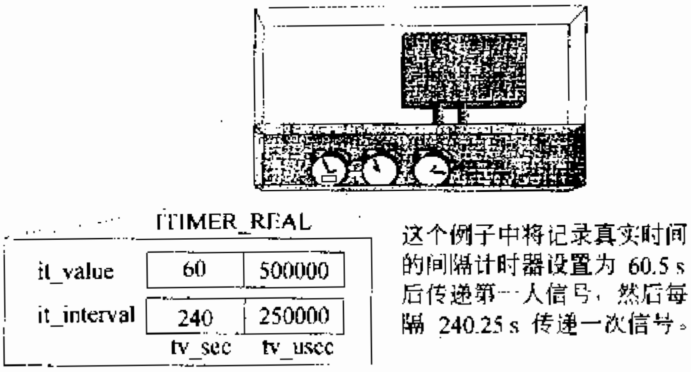


图 7.13　秒和微秒

2. 系统调用小结

getitimer, setitimer	
目标	取得或设置间隔计时器
头文件	#include <sys/time.h>
函数原型	result = getitimer(int which, 　　struct itimerval * val); result = setitimer(int which, 　　const struct itimerval * newval, 　　struct itimerval * oldval);
参数	which　获取或设置的计时器 val　向当前设置值的指针 newval　指向要被设置值的指针 oldval　指向被替换的设置值的指针
返回值	−1　出错 0　成功

getitimer 将某个特定计时器的当前设置读到 val 指向的结构中。setitimer 将计时器设置为 newval 指向的结构的值。如果 oldval 不指向 null，之前计时器的设定将被复制到 oldval 指向的结构中。

which 的值指定将要被读或更新的计时器。计时器的编码分别是 ITIMER_REAL、ITIMER_VIRTUAL 和 ITIMER_PROF。

7.6.5　计算机有几个时钟

如何才能让每个进程有 3 个独立的计时器？有些系统同时有几百个进程在运行。计算机里有几百个独立的时钟吗？不，一个系统只需要一个时钟来设置节拍。这就像用一个节拍器来为一个弦乐四重奏乐团定拍子，或者像有规律摆动的钟摆驱动一个古老钟表里的几根指针。一个硬件时钟的脉冲是计算机里惟一需要的时钟。如何只用一个时钟在设置一个

进程的私有计时器为 5 s 的同时又设置另一个进程的私有计时器为 12 s?

一个古老的时钟是如何让时针、分针和秒针以不同的速度转动的？它们的答案是一样的。每个进程设置自己的计数时间，操作系统在每过一个时间片后为所有的计数器的数值做递减。一个实际的例子可以澄清这些概念。

考虑两个进程：进程 A 和进程 B。进程 A 设置它的真实计时器(real timer)为 5 s，进程 B 设置它的真实计时器为 12 s。为了使数字看起来简单，假设系统时钟每秒跳 100 下。当进程 A 设置它的时钟时，内核设置它的计数器为 500。当进程 B 设置它的时钟时，内核设置它的计数器为 1200，如图 7.14 所示。

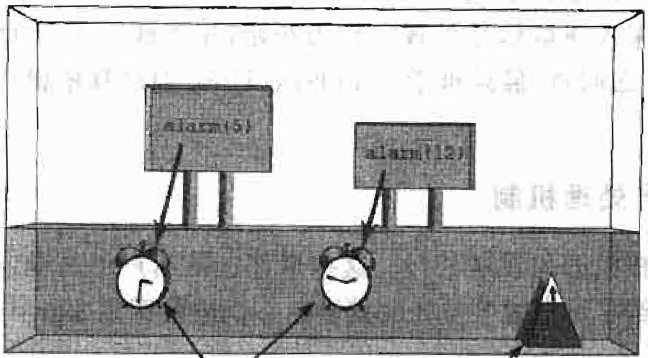


图 7.14 两个计时器、一个时钟

每当内核收到系统时钟脉冲，它遍历所有的间隔计时器，使每个计数器减一个时钟单位。当进程 A 的计数器达到 0 的时候，意味着已经有 500 时钟节拍过去了，内核发送 SIGALRM 给进程 A。如果进程 A 已经设置了计时器的 it_interval 值，内核将这个值复制到 it_value 计数器，否则内核就关掉这个计时器。

再过一会，内核将进程 B 的计数器也减到 0，相应地向进程 B 发出信号。如果 B 设置了计时器的重载值(reload value)，内核就设置 it_value 为相应的值，然后继续处理下一个计时器。

通过这个简单的机制，每个进程就可以设置自己的计时器。这个计时器在进程睡眠的时候也在倒计时。

其他的两个计时器如何工作？它们不是固定的倒计时，而仅仅在进程处于某个特定状态时倒计时。Linux 源代码清楚地给出了它们是如何实现的。

7.6.6 计时器小结

一个 Unix 程序用计时器来挂起执行和调度将要采取的动作。一个计时器是内核的一种机制。通过这种机制，内核在一定的时间之后向进程发送 SIGALRM。alarm 系统调用在特定的实际秒数之后发送 SIGALRM 给进程。setitimer 系统调用以更高的精度控制计时

器，同时能够以固定的时间间隔发送信号。

现在已经知道如何在程序中计时了。要实现视频游戏还需要一项技术：管理中断。

7.7　信号处理 1：使用 signal

这个游戏必须处理中断。可能当用户按下一个键的时候程序正在移动一幅图片，或者在处理一个用户输入的时候，计时器发来一个信号。如果游戏支持双人游戏，当一个人按键的时候程序可能正在处理另一个人的输入。

中断处理是操作系统和系统软件的关键部分。Unix 中的软件中断被称为信号(signals)。现在需要深入理解信号处理。作为开始，先回顾一下早期的 Unix 信号处理模型，然后看看它有些什么问题，最后再学习 POSIX(Unix 型可移植操作系统接口)的信号处理模型。

7.7.1　早期的信号处理机制

各种事件促使内核向进程发送信号。这些事件包括用户的击键、进程的非法操作和计时器到时。第 6 章介绍了早期的信号处理模型。一个进程调用 signal 在以下 3 种处理信号的方法之中选择：

(1) 默认操作(一般是终止进程)，比如，signal(SIGALRM，SIG_DFL)

(2) 忽略信号，比如，signal(SIGALRM，SIG_IGN)

(3) 调用一个函数，比如，signal(SIGALRM，handler)

7.7.2　处理多个信号

如果只有一个信号要处理，原始的信号处理模型足以应付。如果有多个信号到达会发生什么事情？如果响应定义为终止(termination)或者是忽略(ignore)，那结果很清楚。但是如果是调用(invoke)一个函数来响应，那么结果就不那么明显了。

1. 捕鼠器问题

信号处理函数有点像捕鼠器。一个信号意味着什么具有破坏性的事情发生，并被捕获。当信号或老鼠被捕获，信号处理函数或捕鼠器就失效了。

在早期的版本中，信号处理函数在另一个方面也很像捕鼠器：在每次捕获之后，都必须重新设置它们。比如：一个信号处理函数可能如下：

```
void handler(int s)
{
    /*process is vulnerable here*/
    signal(SIGINT,handler);    /*reset handler*/
    ...                        /*do work here*/
}
```

就算设置的速度非常快，它还是需要时间的，在弹簧被触发和设置完成之间，就有可能有老鼠溜走了。这一脆弱的间隙使得原有的信号处理不可靠。有些人称此为“不可靠的信

号”，就好像说“不可靠的老鼠”一样，这倒有些奇怪了。

2. 设计一个更好的系统

捕鼠器问题只是早期信号系统的一个弱点。为了能说明问题的复杂性，考虑以下这些实际生活中的问题。

3. 处理多个信号

真实世界充满信号，也就是意外的打扰。假设你在办公室里工作。电话可能会响，可能有人敲门，或者火警响起。对于这些事件，可以忽略，也可以处理。处理一个电话意味着放下当前的工作，拿起电话，与打电话的人交谈，挂起电话，然后回去做放在一边的工作。处理敲门和这很相似。

如果来访者在你接电话的时候敲门会怎样呢？你得放下电话，按保持键，开门，和来访者交谈，然后回去继续接电话。在接完电话之后，回到办公桌旁继续工作。这种情况下，第二个信号打断了对第一个信号的处理。

接下来，正当你与第一个来访者交谈时，第二个来访者来了又该怎么办呢？一般情况下，第一个人会挡住门，这样第二个人就得等你与第一个人交谈完毕。当你与第一个人交谈完毕，第二个人就可以敲门了。这种情况下，称第二个来访者在接待第一个来访者结束之前被阻塞(blocked)。

还有，如果来访者来访的时候你正专注于电话那头讲话怎么办？当你从门口回来重新拿起电话，是继续刚才的话题还是告诉对方你已经忘记刚刚说到哪儿了？

最后，如果在你处理火警的时候电话响了或者有人敲门又该如何呢？如果一个像火警一样重要的信号达到，你或许希望阻塞其他信号。就像你在处理火警时不会管电话铃是否响了，或者是否有人敲门一样。有些时候就算你没有在处理事件也不想被其他事情打扰。

4. 进程的多个信号

进程要面对的问题和你要面对的没有什么太大不同。想象一个进程在它的小屋(内存)里工作。如图 7.15 所示，用户可能通过按下 Ctrl-C 来产生一个 SIGINT 信号，或者是 Ctrl-\产生 SIGQUIT 信号，或者计时器到时产生一个 SIGQLRM 信号。就像电话和敲门的访者，所有这些信号可能同时到达。在 Unix 系统里，一个进程如何响应多个信号？

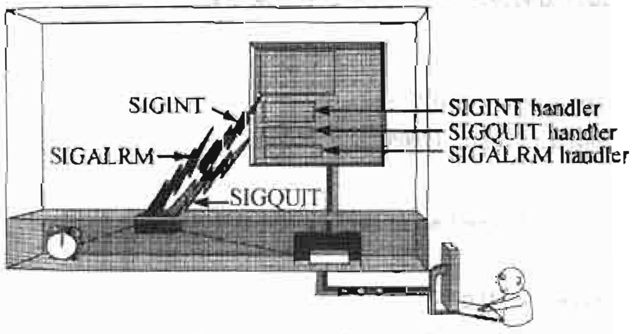


图 7.15　一个接收到多个消息的进程

(1) 处理函数每次使用之后都要被禁用吗？(捕鼠器模型)

(2) 如果 SIGY 消息在进程处理 SIGX 消息时，到达会发生什么？

(3) 如果进程还在处理前一个 SIGX 消息时,第二个 SIGX 消息又到来会发生什么? 第三个又到了呢?

(4) 如果消息到来时,程序正在处理 getchar 或者 read 之类的输入而阻塞,那会如何?

不同版本的 Unix 的答案各不相同。写一个在各个系统下都能正常工作的程序很不容易。

7.7.3 测试多个信号

系统是如何回答这些问题的? 编译并运行 sigdemo3.c,看看你系统中的进程是如何响应信号组合的:

```
/* sigdemo3.c
 * purpose: show answers to signal questions
 * question1: does the handler stay in effect after a signal arrives?
 * question2: what if a signalX arrives while handling signalX?
 * question3: what if a signalX arrives while handling signalY?
 * question4: what happens to read() when a signal arrives?
 */

#include   <stdio.h>
#include   <signal.h>

#define INPUTLEN 100

main(int ac, char * av[])
{
   void   inthandler(int);
   void   quithandler(int);
   char   input[INPUTLEN];
   int    nchars;

   signal( SIGINT, inthandler );        /* set handler */
   signal( SIGQUIT, quithandler );      /* set handler */

   do {
         printf("\nType a message\n");
         nchars = read(0, input, (INPUTLEN - 1));
         if ( nchars ==  -1 )
            perror("read returned an error");
         else {
            input[nchars] = '\0';
            printf("You typed: %s", input);
         }
   }
   while( strncmp( input , "quit", 4 ) != 0 );
}
```

```
void inthandler(int s)
{
    printf(" Received signal %d .. waiting\n", s );
    sleep(2);
    printf(" Leaving inthandler \n");
}

void quithandler(int s)
{
    printf(" Received signal %d .. waiting\n", s );
    sleep(3);
    printf(" Leaving quithandler \n");
}
```

试着以不同的方式常规输入和两个信号生成键 Ctrl−C 和 Ctrl−\。特别地，以不同的时延，试试以下组合跟踪图 7.16 中显示的函数的控制流。

(1) ^C^C^C^C

(2) ^\^C^\^C

(3) hello^C Return

(4) hello Return^C

(5) ^\^\hello^C

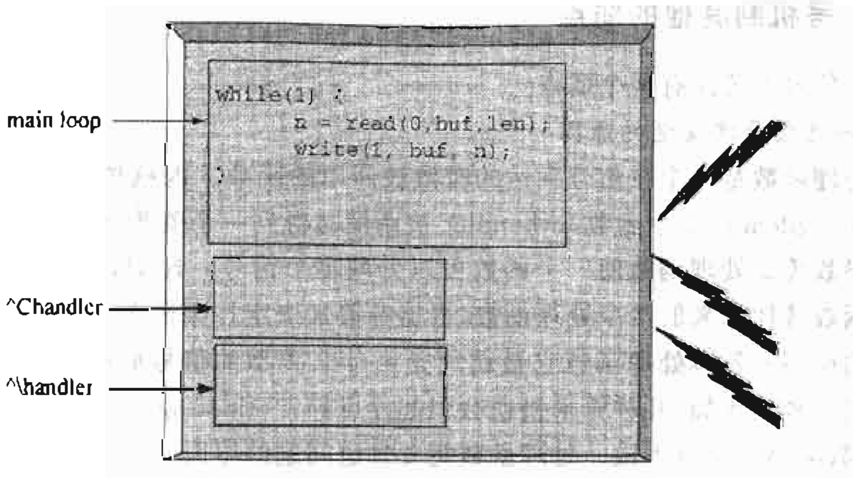


图 7.16　跟踪这些函数的控制流

这些试验的结果显示了你的系统是如何处理信号组合的。

1. 不可靠的信号（捕鼠器）

如果两个 SIGINTS 信号杀死了进程，那么意味着你的系统是不可靠的信号：处理函数必须每次都重置。如果多个 SIGINTS 信号没有杀死进程，意味着处理函数在被调用后还起作用。现代信号处理机制允许你在两者之间做出选择。

2. SIGY 打断 SIGX 的处理函数（接电话的时候有人敲门）

当接连按下 Ctrl−C 和 Ctrl−\会看到程序先跳到 inthandler，接着跳到 quithandler，然后

再回到 inthandler，最后回到主循环。你的试验结果是如何的？

3. SIGX 打断 SIGX 的处理函数（两次敲门）

这种情况就像两个人来敲门。有 3 种处理的方法：

(1) 递归，调用同一个处理函数[①]；

(2) 忽略第二个信号，这和没有呼叫等待功能的电话机类似；

(3) 阻塞第二个信号直到第一个处理完毕。

原来的信号处理系统使用方法(1)，允许递归调用。一个更安全的选择是方法(3)。就像第二个人来敲门，第二个信号被阻塞而不是忽略，直到第一个信号被处理完。你的系统阻塞第二个信号吗？还是递归调用处理函数？你的系统将多个信号排队吗？

4. 被中断的系统调用（接电话的时候有人敲门）

这种情况经常发生。程序经常在等待输入的时候接收到信号。在上面那个测试程序中，主循环阻塞以等待键盘的输入（read 调用）。当输入中断（Ctrl－C）或退出（Ctrl－\）键，程序跳转到信号处理函数。当处理函数处理完成后，程序回到主循环，应该回到跳离的位置。但果真如此吗？如果输入"hel"，接着按下 Ctrl－C 然后再继续输入"lo"再回车会如何呢？程序看到的是完整的"hello"还是只有"lo"？程序是重新开始 read 还是从 read 返回同时设置 errno 到 EINTR？

是重新开始还是返回呢？在 AT&T 的 Unix 中是返回并设置 EINTR 为－1（经典模式），而在 UCB 中会自动的重新开始。

7.7.4 信号机制其他的弱点

早期的信号系统还有两个弱点。

1. 不知道信号被发送的原因

信号处理函数是一个在信号到达的时候被调用的函数。内核传给处理函数一个信号数字编号。在 sigdemo3.c 中函数 inthandler 被调用时得到一个值为 SIGINT 的实参。将信号编号作为参数传给处理函数使一个函数可以处理多个信号。比如，在 sigdemo3.c 中可以用一个处理函数替代原来的两个处理函数，根据参数来决定打印什么。

早期的模型只告诉处理函数它被调用是由什么类型的信号而引起的，但是没有告之为什么会生成信号。比如，几种算术错误会引起浮点异常（floating－point exception）。比如除数为零、整数溢出和浮点下溢。处理函数需要知道问题的原因。

2. 处理函数中不能安全地阻塞其他消息

响应一个火警时，一般情况下会忽略电话铃声。假设想让程序在响应 SIGINT 时忽略 SIGQUIT。使用经典信号机制修改 inthandler，如下所示：

```
void inthandler(int s)
{
    int rv;
    void ( * prev_qhandler)();          /*holds prev handler*/
```

① 这是非显式递归，因为处理函数并没有调用自己，但其效果同普通的递归相似。

```
    prev_qhandler = signal(SIGQUIT,SIG_IGN);
                                        /*ignore QUITs*/
    ...
    signal(SIGQUIT,prev_qhandler);      /*restore handler*/
}
```

这样，在进入中断处理函数时禁用退出处理函数，在结束时再重新使能它。这个方案有两个问题。第一在调用 inthandler 和调用 signal 之间是它的软肋所在。这里只是希望调用 inthandler 和忽略 SIGQUIT 同时进行。第二，这里并不想忽略 SIGQUIT，而只是想阻塞它直到 inthandler 处理完成，如同想在火警之后再回来接电话。在处理完关键事件后，还是很高兴看到 SIGQUIT 回来工作。

7.8　信号处理 2：sigaction

在过去的几年中，针对原来的模型所产生的问题，各种相关组织开发出了一些不同的解决方案。这里将只学习 POSIX 模型和相关的系统调用。经典的信号系统依旧被支持，而且在一些应用中这些就够了。

7.8.1　处理多个信号：sigaction

在 POSIX 中用 sigaction 替代 signal。参数非常相似。指定什么信号将被如何处理。如果愿意，还能得到这个信号上一次被处理时的设置。

int sigaction(signalnumber, action, prevaction)

这个函数的概要如下。

sigaction	
目标	指定一个信号的处理函数
头文件	#include <signal.h>
函数原型	res = sigaction(int signum, const struct sigaction * action, struct sigaction * prevaction);
参数	signum　要处理的信号 action　指针，指向描述操作的结构 prevaction　指针，指向描述被替换操作的结构
返回值	-1　失败 0　成功

第一个参数 signum 指明想要处理的消息。第二个参数 action 指向描述如何响应信号的结构体。第三个参数 prevaction 如果不是 null 的话，就是指向描述被替换的处理设置的结构体。如果新的操作设置成功则返回 0，否则返回 -1。

1．**定制信号处理：struct sigaction**

在过去，面对信号的处理只有简单的几种选择：SIG_DFL、SIG_IGN 或者函数处理。这

些选项在新的系统中作为结构体 sigaction 的部分定义依然提供。结构体 sigaction 定义了如何处理一个信号。以下是这个结构体的完整定义：

```
struct sigaction{
    /* sc only one of these two */
    void ( * sa_handler)(); /*SIG_DFL,SIG_IGN,or function */
    void ( * sa_sigaction)(int,siginfo_t * ,void * ); /* new handler */

    sigset_t sa_mask;          /* signals to block while handling */
    int sa_flags;              /* enable various behaviors */
}
```

(1) 选择 sa_handler 还是 sa_sigaction?

首先，要在老的信号处理方式和新的更强大的信号处理方式之间作出选择。如果老的处理方式(即 SIG_DFL、SIG_IGN 或者处理函数)就够用了，那么可以设置 sa_handler 为其中之一。当然，如果指定为旧的信号处理方式，那么只能得到信号编号。否则，如果设定 sa_sigaction 为一个处理函数，那么那个处理函数被调用的时候，不但可以得到信号编号而且可以获悉被调用的原因以及产生问题的上下文的相关信息。两者之间的差异总结如下。

使用旧的处理机制：	使用新的处理机制：
struct sigaction action;	action.sa_handler = handler_old;
struct sigaction action;	action.sa_sigaction = handler_new;

如何告诉内核你使用的是新的信号处理方式？很简单，只需设置 sa_flags 的 SA_SIGINFO 位。

(2) sa_flags

然后，决定处理函数将如何应对前面提出的 4 个问题。sa_flags 是用一些位来控制处理函数如何应对上述 4 个问题的。相关的细节可以在手册上查到。下面是部分列表。

标　记	含　义
SA_RESETHAND	当处理函数被调用时重置。也就是采用捕鼠器模式
SA_NODEFER	在处理信号时关闭信号自动阻塞。这样就允许递归调用信号处理函数
SA_RESTART	当系统调用是针对一些慢速的设备或类似的系统调用，重新开始，而不是返回。这样是采用 BSD 模式
SA_SIGINFO	指明使用 sa_sigaction 的处理函数的值。如果这个位没有被设置，那么就使用 sa_handler 指向的处理函数的值。如果 sa_sigaction 被使用，传给处理函数将不只是信号编号，还包括指向描述信号产生的原因和条件的结构体

(3) sa_mask

最后，决定在处理一个消息时是否要阻塞其他信号。sa_mask 中的位指定哪些信号要被阻塞。使用 sa_mask，可以在逃离火灾现场时阻塞电话呼叫和来访者。sa_mask 的值包括要

被阻塞的信号集。阻塞信号是防止数据损毁的重要技术。下一章将详细介绍这个主题。

2. 例子：使用 sigaction

下面的例子演示了如何使用 sigaction（注意程序如何做到在处理 SIGINT 时阻塞 SIGQUIT 的）：

```
/* sigactdemo.c
 *            purpose: shows use of sigaction()
 *            feature : blocks ^\ while handling ^C
 *                    does not reset ^C handler, so two kill
 */

#include  <stdio.h>
#include  <signal.h>
#define   INPUTLEN   100

main()
{
   struct sigaction newhandler;         /* new settings */
   sigset_t         blocked;            /* set of blocked sigs */
   void             inthandler();       /* the handler */
   char             x[INPUTLEN];

   /* load these two members first */
   newhandler.sa_handler = inthandler;
                                        /* handler function */
   newhandler.sa_flags = SA_RESETHAND | SA_RESTART;
                                        /* options */

   /* then build the list of blocked signals */
   sigemptyset(&blocked);               /* clear all bits */
   sigaddset(&blocked, SIGQUIT);        /* add SIGQUIT to list */
   newhandler.sa_mask = blocked;        /* store blockmask */

   if ( sigaction(SIGINT, &newhandler, NULL) == -1 )
      perror("sigaction");
   else
      while( 1 ){
         fgets(x, INPUTLEN, stdin);
         printf("input: %s", x);
      }
}

void inthandler(int s)
{
   printf("Called with signal %d\n", s);
   sleep(s);
```

```
    printf("done handling signal %d\n", s);
}
```

试着运行这个程序。如果以很快的速度连续按 Ctrl-C 和 Ctrl-\，退出信号将被阻塞直到中断信号处理完毕。如果连续按两下 Ctrl-C，进程就将被第二个信号杀死。如果想要捕获所有的 Ctrl-C，将 SA_RESETHAND 掩码从 sa_flags 中去掉。

7.8.2 信号小结

一个进程可能被各种来源的信号中断。信号可能在任何时候以任何顺序到达。signal 提供了一种简单但是不完整的信号处理机制。POSIX 接口，即 sigaction 提供了复杂的、明确定义的方法来控制进程如何对各种信号组合做出反应。

现在已经知道如何在程序中管理时间和中断。视频游戏还需要最后一项技术：防止混乱。

7.9 防止数据损毁(Data Corruption)

你有曾经被同时做几件事情搞得晕头转向并因此犯错误的经历吗？如果门铃在你寻找邮票的时候响起，你可能会寄出一封没有贴邮票的信。程序也有同样的问题。当它们正在处理一件事情时被调用到其他地方，它们就可能会被搞晕以至于把事情弄得一团糟。

来看看在现实生活中中断是如何引起数据错误的。然后学习在程序中如何防止这种情况发生。

7.9.1 数据损毁的例子

继续那个办公室的例子。敲你办公室门的人要把他的名字和地址写到一个列表里。每个人只在列表的最后添加一项记录：姓名、街道、城市、省份和邮编。考虑以下两个问题。

第一，当一个人正在往列表里写信息时有人打电话来要列表里的名字和地址。如果你将列表的信息读给人家，就会有给出不完整信息的可能。这可以通过在受访时挂掉电话来阻止这类错误的发生。

第二，考虑一个不同的问题。当一个访问者刚刚把姓名填好，第二个 SIGKNOCK 信号到达。如果允许递归的信号处理，第一个访者要等第二个访者填好他的信息后再继续在列表的末尾填自己余下的信息。这样列表就包含了错误的数据：一条记录在另外一条记录中间。这可以通过一个接一个而不是递归的接待访者来防止这类错误。

这两个例子说明了在一些情况下一个操作不应该被其他操作打断。在对一个数据结构(这里是列表)改动结束之前，其他函数不能读或写这个数据结构。当然，处理像火警一类的信号是安全的，因为这类处理并不读或修改数据。

7.9.2 临界区(Critical Sections)

一段修改一个数据结构的代码如果在运行时被打断将导致数据的不完整或损毁，则称

这段代码为临界区。当程序处理信号时，必须决定哪一段代码为临界区，然后设法保护这段代码。临界区不一定就在信号处理函数中，很多出现在常规的程序流中。保护临界区的最简单的办法就是阻塞或忽略那些处理函数将要使用或修改特定数据的信号。

7.9.3　阻塞信号：sigprocmask 和 sigsetops

可以在信号处理者一级或进程一级阻塞信号。

1. 在信号处理者一级阻塞信号

为了在处理一个信号的时候阻塞另一个信号，要设置 struct sigaction 结构中的 sa_mask 成员位，它在设置处理函数时被传递给 sigaction。sa_mask 是 sigset_t 类型，它定义了一个信号集。我们将简要地解释这个集合。

2. 一个进程的阻塞信号

在任何时候一个进程都有一些信号被阻塞。注意，是阻塞而不是忽略。这个信号集就称为信号挡板（signal mask）。通过 sigprocmask 可以修改这个被阻塞的信号集，sigprocmask 作为一个原子操作根据所给的信号集来修改当前被阻塞的信号集：

sigprocmask	
目标	修改当前的信号挡板
头文件	#include <signal.h>
函数原型	int res = sigprocmask(int how, const sigset_t * sigs, sigset_t * prev);
参数	how　如何修改信号挡板 sigs　指向使用的信号列表的指针 prev　指向之前的信号挡板列表的指针（或为 null）
返回值	−1　失败 0　成功

sigprocmask 修改当前的信号挡板设置。当 how 的值分别为 SIG_BLOCK、SIG_UNBLOCK 或 SIG_SET 时，* sigs 所指定的信号将被添加、删除或替换。如果 prev 不是 null，那么之前的信号挡板设置将被复制到 * prev 中。

3. 用 sigsetops 构造信号集

一个 sigset_t 是一个抽象的信号集，可以通过一些函数来添加或删除信号。基本的函数如下：

```
sigemptyset(sigset_t * setp)
```

清除由 setp 指向的列表中的所有信号。

```
sigfillset(sigset_t * setp)
```

添加所有的信号到 setp 指向的列表。

```
sigaddset(sigset_t * setp, int signum)
```

添加 signum 到 setp 指向的列表。

sigdelset(sigset_t * setp, int signum)

从 setp 指向的列表中删除 signum 所标识的信号。

4. 例子：暂时地阻塞用户信号

程序可以用以下代码来暂时地阻塞 SIGINT 和 SIGQUIT 信号：

```
sigset_t sigs,prevsigs;                    /* define two signal sets */
sigemptyset(&sigs);                         /* turn off all bits */
sigaddset(&sigs, SIGINT);                   /* turn on SIGINT bit */
sigaddset(&sigs, SIGQUIT);                  /* turn on SIGQUIT bit */
sigprocmask(SIG_BLOCK, &sigs, &prevsigs);   /* add that to proc mask */
// .. modify data structure here..
sigprocmask(SIG_SET, * prevsigs, NULL);     /* restore previous mask */
```

注意这里是如何在修改一个 tty 驱动或者文件描述符的时候保存先前的设置，然后用保存的设置来恢复原来的信号挡板。除非目的就是修改获取的资源，否则释放资源时恢复获取时的状态是个好习惯。

7.9.4 重入代码(Reentrant Code)：递归调用的危险

打断别人登记，而将自己的名字和地址插在别人的记录中间的例子引入另一个与数据损毁有关的概念：可重入函数。

一个信号处理者或者一个函数，如果在激活状态下能被调用而不引起任何问题就称之为可重入的。

在通过 sigaction 设置时，可以通过设置 SA_NODEFER 位来允许处理函数的递归调用。反之，可以通过清除此位来阻塞信号。如何选择呢？

如果处理者不是可重入的，必须阻塞信号。但是如果阻塞信号，就有可能丢失信号。信号不像电话便条那样贴在那里等你回来处理。有些信号是非常重要的：丢掉一些是安全的吗？

是丢掉信号还是弄乱数据？哪个更糟些？有没有办法同时避免这两个问题？设计使用信号的程序时，这些是必须考虑的问题。信号处理上的错误现象不很有规律，尤其在系统处于高负载的情况下或者在精确的性能计量的时候。排错需要理解信号处理的工作机理，还要知道哪里可能会有问题。

7.9.5 视频游戏中的临界区

球匀速在屏幕上移动，碰到墙或挡板就弹回。用户通过按键上下移动挡板。间隔计数器控制球的运动。用户控制挡板的输入就如信号那样看上去是无法预料的事件。需要在某个时刻阻塞用户输入吗？有没有什么临界区，其间挡板不应该移动吗？在应用所有已学的知识来完成视频游戏之前，先来看看另一个信号的来源：其他进程。

7.10　kill：从另一个进程发送的信号

信号来自间隔计时器、终端驱动、内核或者进程。一个进程可以通过 kill 系统调用向另一个进程发送信号：

kill		
目标	向一个进程发送一个信号	
头文件	#include <sys/types.h> #include <signal.h>	
函数原型	int kill(pid_t pid, int sig)	
参数	pid sig	目标进程 id 要被发送的信号
返回值	−1 0	失败 成功

kill 向一个进程发送一个信号。发送信号的进程的用户 ID 必须和目标进程的用户 ID 相同，或者发送信号的进程的拥有者是一个超级用户。一个进程可以向自己发送信号。

一个进程可以向其他进程发送任何信息，包括一般来自键盘、间隔计时器或者内核的信号。比如一个进程可以向另一个进程发送 SIGSEGV 信号，就好像目标进程执行了非法内存读取。

Unix 命令 kill 使用 kill 系统调用（如图 7.17 所示）。

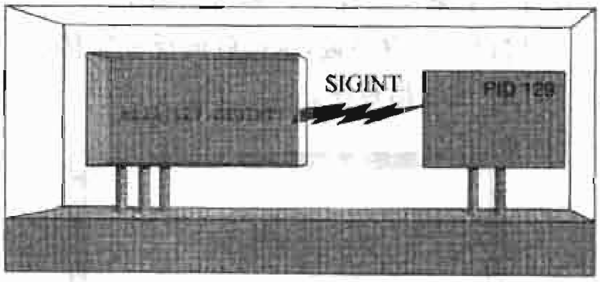


图 7.17　一个进程使用 kill()来发送信息

1. 进程间通信的含义

接受信号的进程几乎可以设置任何信号的处理者。考虑一下在收到 SIGINT 时就打印 OUCH！的程序。如果其他进程向 OUCH！程序发送 SIGINT 又该如何呢？OUCH！程序会捕获信号，跳转到处理者，打印 OUCH！（如图 7.18 所示）。

更进一步。如果第一个程序设置一个间隔计时器，计时器的信号处理函数向 OUCH！程序发送 SIGINT 信号。这样相应的处理函数就被调用。从而一个进程的计时器控制了另一个进程的函数调用。实际上，一组进程可以像橄榄球运动员传递橄榄球那样传递信号。

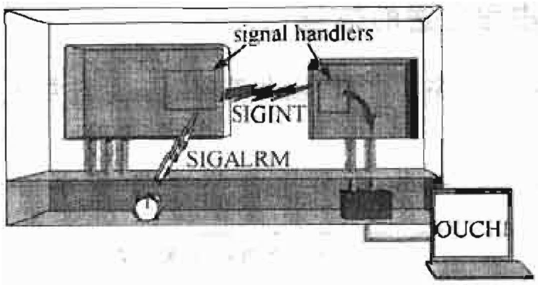


图 7.18　信号的复杂用法

2. IPC 信号设计：SIGUSR1、SIGUSR2

Unix 有两个信号可以被用户程序使用，它们是 SIGUSR1 和 SIGUSR2。这两个信号没有预定义任务。可以使用它们以避免使用已经有预定义语义的信号。

将在后面几章学习进程间通信。编程时可以有很多方法组合使用 kill 和 sigaction。

7.11　使用计时器和信号：视频游戏

现在回到视频游戏。游戏有两个主要元素：动画和用户输入。动画要平滑，用户输入会改变运动状态。下一个程序 bounce1d.c 让用户可以将字符串在屏幕上弹来弹去。

7.11.1　bounce1d.c：在一条线上控制动画

首先来看看 bounce1d 看上去是什么样子。界面如图 7.19 所示。bounce1d.c 将一个单词平滑地在屏幕上移动。当用户按下空格键，单词就向反方向移动。"s"键和"f"键分别增加和减少单词的移动速度。按"Q"键退出程序。

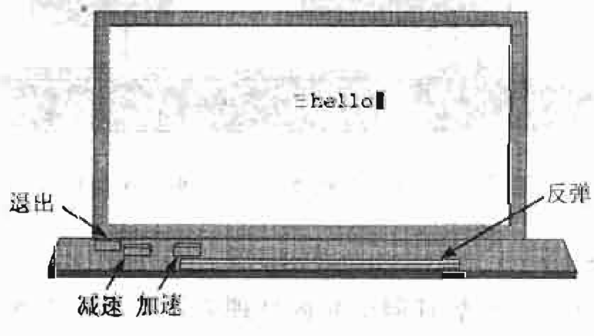


图 7.19　bounce1d 的运行界面：用户控制的动画

这个程序是如何实现的呢？我们已经知道如何实现动画。在一个地方画一个字符串，等几毫秒，然后擦去旧的影像并在原来位置的左边或右边一个单位距离重新画同一个字符串。这里希望擦去和重画动作以相同的间隔连续的进行。所以使用间隔计时器来调用相应的处理函数。

两个变量分别记录移动的方向和速度。设置方向变量的值为＋1 和 －1 分别表示向左

和向右移动。延时变量记录间隔计时器的间隔长度。较长的延时意味着较慢的速度，反之则意味着较快的速度。

现在向程序添加方向和速度控制。根据用户的键盘输入修改方向和速度变量。程序的逻辑如图 7.20 所示。bounce1d 体现了两个重要的技术：状态变量和事件处理。记录位置、方向和延时的变量定义了动画的状态。用户输入和计时器信号是改变这些状态的事件。每次计时器到达信号就调用改变位置的处理函数。每次得到用户键盘输入信号就调用改变方向和速度变量的代码。以下是它的代码。

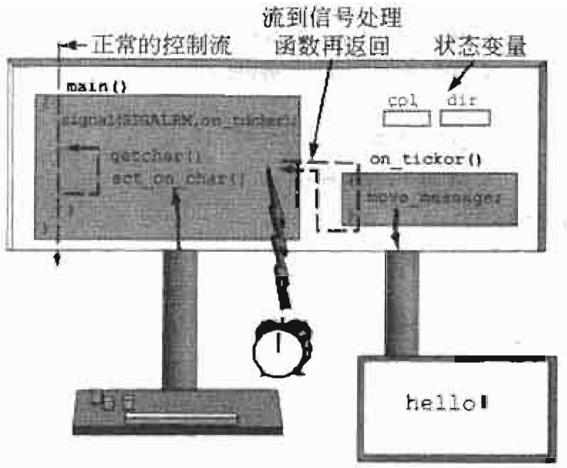


图 7.20　用户输入改变变量值而变量值控制动作

```
/* bounce1d.c
 * purpose        animation with user controlled speed and direction
 * note           the handler does the animation
 *                the main program reads keyboard input
 * compile        cc bounce1d.c set_ticker.c -l curses -o bounce1d
 */
#include  <stdio.h>
#include  <curses.h>
#include  <signal.h>

/* some global settings main and the handler use */

#define  MESSAGE   "hello"
#define  BLANK     "     "

int row;   /* current row     */
int col;   /* current column  */
int dir;   /* where we are going */

int main()
{
    int delay;                /* bigger => slower */
    int ndelay;               /* new delay */
```

```
    int c;                   /* user input */
    void    move_msg(int);   /* handler for timer */

    initscr();
    crmode();
    noecho();
    clear();

    row = 10;                /* start here */
    col = 0;
    dir = 1;                 /* add 1 to row number */
    delay = 200;             /* 200ms = 0.2 seconds */

    move(row,col);           /* get into position */
    addstr(MESSAGE);         /* draw message */
    signal(SIGALRM, move_msg );
    set_ticker( delay );

    while(1)
    {
       ndelay = 0;
       c = getch();
       if ( c == 'Q' ) break;
       if ( c == ' ' ) dir = -dir;
       if ( c == 'f' && delay > 2 ) ndelay = delay/2;
       if ( c == 's' ) ndelay = delay * 2 ;
       if ( ndelay > 0 )
          set_ticker( delay = ndelay );
    }
    endwin();
    return 0;
}

void move_msg(int signum)
{
    signal(SIGALRM, move_msg);   /* reset, just in case */
    move( row, col );
    addstr( BLANK );
    col + = dir;                 /* move to new column */
    move( row, col );            /* then set cursor */
    addstr( MESSAGE );           /* redo message */
    refresh();                   /* and show it */

    /*
     * now handle borders
     */
    if ( dir == -1 && col <= 0 )
```

```
        dir = 1;
    else if ( dir == 1 && col + strlen(MESSAGE) >= COLS )
        dir = -1;
}
```

1. 递归还是阻塞：一个真实的例子

在学习信号处理函数的数据损毁时提到过重入函数。bounce1d 提供了一个考察这个问题的真实例子。开始时信号处理函数 move_msg 每秒钟被调用 5 次。按"f"键来减小计时器延时以增加动画速度。如果按很多次"f"键，两次计时器消息之间的间隔可能比一次处理函数的执行时间还要短。如果计时器消息在处理函数忙于擦去和重画字符串时到达又会如何？

这个问题的分析留作习题。在这个程序中使用 signal，到底是递归还是阻塞依赖于你的系统。

2. 下一步做什么？

如何扩展 bounce1d 为一个弹球游戏？首先，要用"O"来替换"hello"，因为"O"更像一个球。然后，要让球在左右移动之外还可上下移动。为了增加上下移动的能力要添加状态变量。现在已经有 col 和 row 来记录球的位置，dir 来记录水平移动方向。如果要使球能上下移动，还要添加什么变量呢？

7.11.2　bounce2d.c：两维动画

程序 bounce2d 产生两维的动画，可以让用户控制水平速度和垂直速度，如图 7.21 所示。

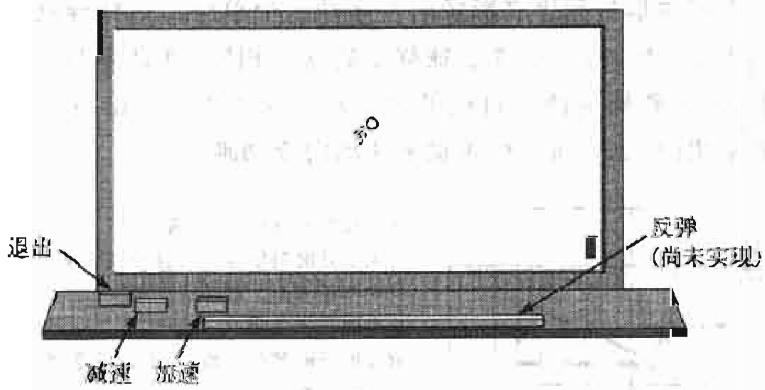


图 7.21　两维动画

bounce2d 的 3 个设计部分与 bounce1d 相同。

(1) 计时器驱动

间隔计时器被设置为产生固定的 SIGALRMS 信号流。响应一个信号，球向前移动一步。

(2) 等待键盘输入

程序阻塞等待键盘输入。根据用户按下的键的不同采取不同的动作。

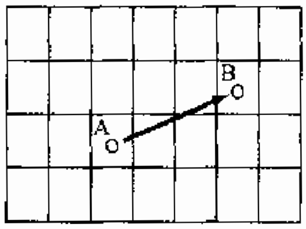
(3) 状态变量

变量记录了球的速度和方向。用户输入修改的变量值决定了小球的速度。计时器处理

函数根据速度和位置变量来决定在何时何处画小球。

看起来都很像 bounce1d。但这里有一点不同，这是一个重要的问题：如何让球斜着移动？

让球斜着移动是一个新问题。在一维的程序中，每个计时器信号促使影像在屏幕上移动一个单位。这样就很简单：一个信号、一个单位。但是在两维的动画中，情况就不这么简单了。考虑一下图 7.22 所示的路径。这种情况下，每垂直移动一个单位，水平移动 3 个单位。一种方法是在收到每个计时器信号时将影像从 A 移动到 B。当两个直角边成一定比例时，影像的跳跃可能比较大。比如当比例为 3/4 时每次跳跃达到 5 个单位。

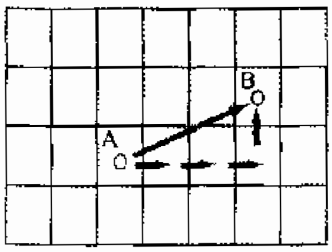


问题：如何将"O"平滑的从单元 A 移动到单元 B?

图 7.22　直角边之比为 1/3 的移动路径

每次移动一个单位看起来比较好。当直角边的比为 1/3 的时候，影像如图 7.23 那样一个单位一个单位的移动。注意，影像每横向移动 3 个单位纵向就移动一个单位。横向移动的步数是纵向移动步数的 3 倍。

这样看起来像有两个计时器。考虑只有一个计时器。每隔两个时钟信号影像向右移动一个单位。每隔 6 个时钟信号程序将影像向上移动一个单位。这样球就会斜着移动。如果时钟间隔单位分别为 10 和 30 的话，那么球移动的路径相同，就是慢些。一个程序只有一个真实的间隔计时器，所以要构造两个自己的计时器。这两个自己的计时器是由真实的间隔计时器驱动。这里采用图 7.23 所示的逻辑来驱动两维动画。



为了模拟对角线移动，需要：
每两个计时器信号向右移动一个单位，每六个计时器信号向上移动一个单位。
这种的技术需要两个计时器。一个记录水平移动的计时器信号数，一个记录垂直移动的计时器信号数。

图 7.23　每次移动一个单位更好

为了能同时在两个方向移动，用两个计数器来充当计时器。就像系统的间隔计时器由两部分组成一样，每个计数器都有两个属性：当前值和间隔。当前值表示在下次重画之前还要等待多少个计时信号。间隔指明两次移动之间所要等待的计时信号个数。两个成员变量分别命名为 ttg 和 ttm。代码如下：

```
/* bounce2d 1.0
 * bounce a character (default is 'o') around the screen
```

```
 * defined by some parameters
 *
 * user input: s slow down x component, S: slow y component
 *         f speed up x component, F: speed y component
 *         Q quit
 *
 * blocks on read, but timer tick sends SIGALRM caught by ball_move
 * build: cc bounce2d.c set_ticker.c -lcurses -o bounce2d
 */
#include   <curses.h>
#include   <signal.h>
#include   "bounce.h"

struct ppball the_ball ;

/** the main loop **/

void set_up();
void wrap_up();

int main()
{
    int c;

    set_up();

    while ( ( c = getchar()) != 'Q' ){
        if ( c == 'f' )        the_ball.x_ttm-- ;
        else if ( c == 's' ) the_ball.x_ttm++ ;
        else if ( c == 'F' ) the_ball.y_ttm-- ;
        else if ( c == 'S' ) the_ball.y_ttm++ ;
    }

    wrap_up();
}

void set_up()
/*
 * init structure and other stuff
 */
{
    void ball_move(int);

    the_ball.y_pos = Y_INIT;
    the_ball.x_pos = X_INIT;
    the_ball.y_ttg = the_ball.y_ttm = Y_TTM ;
    the_ball.x_ttg = the_ball.x_ttm = X_TTM ;
    the_ball.y_dir = 1 ;
```

```
    the_ball.x_dir = 1 ;
    the_ball.symbol = DFL_SYMBOL ;

    initscr();
    noecho();
    crmode();

    signal( SIGINT , SIG_IGN );
    mvaddch( the_ball.y_pos, the_ball.x_pos, the_ball.symbol );
    refresh();

    signal( SIGALRM, ball_move );
    set_ticker( 1000 / TICKS_PER_SEC );
        /* send millisecs per tick */
}

void wrap_up()
{

    set_ticker( 0 );
    endwin();                               /* put back to normal */
}

void ball_move(int signum)
{
    int   y_cur, x_cur, moved;

    signal( SIGALRM , SIG_IGN );            /* dont get caught now */
    y_cur = the_ball.y_pos ;                /* old spot */
    x_cur = the_ball.x_pos ;
    moved = 0 ;

    if ( the_ball.y_ttm > 0 && the_ball.y_ttg-- == 1 ){
       the_ball.y_pos += the_ball.y_dir ;   /* move */
       the_ball.y_ttg = the_ball.y_ttm ;    /* reset */
       moved = 1;
    }

    if ( the_ball.x_ttm > 0 && the_ball.x_ttg-- == 1 ){
       the_ball.x_pos += the_ball.x_dir ;   /* move */
       the_ball.x_ttg = the_ball.x_ttm ;    /* reset */
       moved = 1;
    }

    if ( moved ){
       mvaddch( y_cur, x_cur, BLANK );
       mvaddch( y_cur, x_cur, BLANK );
       mvaddch( the_ball.y_pos, the_ball.x_pos, the_ball.symbol );
       bounce_or_lose( &the_ball );
```

```
        move(LINES - 1,COLS - 1);
        refresh();
    }
    signal( SIGALRM, ball_move);       /* for unreliable systems */

}

int bounce_or_lose(struct ppball * bp)
{
    int return_val = 0 ;

    if ( bp->y_pos == TOP_ROW ){
        bp->y_dir = 1 ;
        return_val = 1 ;
    } else if ( bp->y_pos == BOT_ROW ){
                bp->y_dir = -1 ;
                return_val = 1;
    }
    if ( bp->x_pos == LEFT_EDGE ){
        bp->x_dir = 1 ;
        return_val = 1 ;
    } else if ( bp->x_pos == RIGHT_EDGE ){
                bp->x_dir = -1;
                return_val = 1;
    }

    return return_val;
}
```

头文件为：

```
/* bounce.h               */

/* some settings for the game */

#define   BLANK        ' '
#define   DFL_SYMBOL    'o'
#define   TOP_ROW      5
#define   BOT_ROW      20
#define   LEFT_EDGE     10
#define   RIGHT_EDGE     70
#define   X_INIT     10    /* starting col    */
#define   Y_INIT     10    /* starting row    */
#define   TICKS_PER_SEC    50     /* affects speed */

#define   X_TTM     5
#define   Y_TTM     8
```

```
/** the ping pong ball **/

struct ppball {
        int y_pos, x_pos,
            y_ttm, x_ttm,
            y_ttg, x_ttg,
            y_dir, x_dir;
        char symbol ;

    } ;
```

7.11.3　完成游戏

剩下的工作作为练习留给读者。需要添加挡板控制球的代码,将球从挡板弹回的代码,判断球是否已经飞出界的代码等。

到在为止已经学习了所有实现游戏所需要的技能。小心斟酌代码的重入问题。什么地方的代码会在一个时间多次被调用?想让计时器处理函数设计成递归调用的,还是阻塞后面的信号?

7.12　输入信号:异步 I/O

本章的动画和游戏等待两类事件:计时器信号和键盘输入。设置间隔计时器的处理函数来控制动画,通过调用 getch 阻塞程序以等待键盘输入。除了阻塞,还能像得到计时器信号那样通过信号来得到用户的输入吗?

可以的。程序可以要求内核在得到输入时发送信号。这有点像要求邮递员在投递邮件时按门铃。这样你就不用坐在大门前整天盯着邮件箱而干些其他什么事情或者睡觉。任何时候只要一有信到你就能知道。

Unix 有两个异步输入(asynchronous input)系统。一种方法是当输入就绪时发送信号,另一个系统当输入被读入时发送信号。UCB 中通过设置文件描述块(file descriptor)的 O_ASYNC 位来实现第一种方法。第二种方法是 POSIX 标准,它调用 aio_read。下面将学习这两种方法。首先,来看看这么做的想法。

7.12.1　使用异步 I/O

新版本的反弹程序如图 7.24 所示。需要两种信号:SIGIO 和 SIGALRM,所以要建立两个处理函数。SIGIO 处理函数读入击键并根据读入的数据采取行动。SIGALRM 处理函数驱动动画并检测碰撞。为简单起见,去掉了速度控制。

7.12.2　方法 1:使用 O_ASYNC

使用 O_ASYNC 需要对原有的弹球程序做 4 处改动。首先要建立和设置在键盘输入时被调用的处理函数。其次,使用 fcntl 的 F_SETOWN 命令来告诉内核发送输入通知信号给进程。其他进程可能也连接到键盘。这里不想让这些进程发送信号。第三,通过调用 fcntl

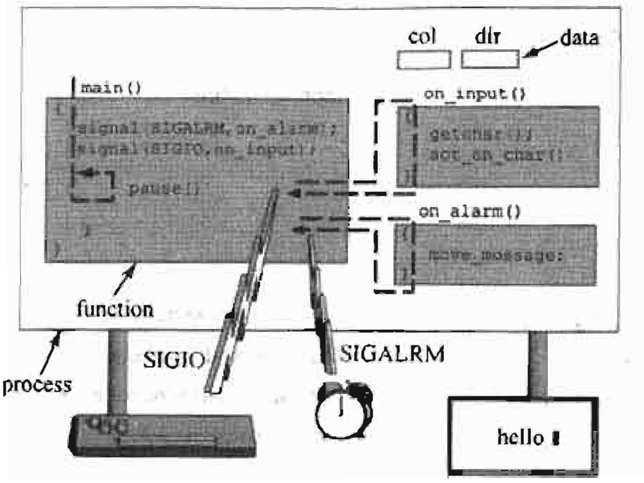


图 7.24 键盘和计时器都发送信号

来设置文件描述符 0 中的 O_ASYNC 位来打开输入信号。最后，循环调用 pause 等待来自计时器或键盘的信号。当有一个从键盘来的字符到达，内核向进程发送 SIGIO 信号。SIGIO 的处理函数使用标准的 curses 函数 getch 来读入这个字符。当计时器间隔超时，内核发送以前已经处理的 SIGALRM 信号。以下是源代码：

```
/* bounce_async.c
 * purpose    animation with user control, using O_ASYNC on fd
 * note       set_ticker() sends SIGALRM, handler does animation
 *            keyboard sends SIGIO, main only calls pause()
 * compile    cc bounce_async.c set_ticker.c -l curses -o bounce_async
 */
#include   <stdio.h>
#include   <curses.h>
#include   <signal.h>
#include   <fcntl.h>

/* The state of the game */

#define   MESSAGE   "hello"
#define   BLANK     "     "

int row   = 10;                  /* current row */
int col   = 0;                   /* current column   */
int dir   = 1;                   /* where we are going */
int delay = 200;                 /* how long to wait */
int done  = 0;

main()
{
    void   on_alarm(int);           /* handler for alarm */
```

```
    void   on_input(int);              /* handler for keybd */
    void   enable_kbd_signals();

    initscr();                         /* set up screen */
    crmode();
    noecho();
    clear();


    signal(SIGIO, on_input);           /* install a handler */
    enable_kbd_signals();              /* turn on kbd signals */
    signal(SIGALRM, on_alarm);         /* install alarm handler */
    set_ticker(delay);                 /* start ticking */

    move(row,col);                     /* get into position */
    addstr( MESSAGE );                 /* draw initial image */

    while( ! done )                    /* the main loop */
        pause();
    endwin();
}
void on_input(int signum)
{
    int c = getch();                   /* grab the char */

    if ( c == 'Q' || c == EOF )
        done = 1;
    else if ( c == ' ' )
        dir = -dir;
}

void on_alarm(int signum)
{
    signal(SIGALRM, on_alarm);         /* reset, just in case */
    mvaddstr( row, col, BLANK );       /* note mvaddstr() */
    col += dir;                        /* move to new column */
    mvaddstr( row, col, MESSAGE );     /* redo message */
    refresh();                         /* and show it */

    /*
     * now handle borders
     */
    if ( dir == -1 && col <= 0 )
        dir = 1;
    else if ( dir == 1 && col + strlen(MESSAGE) >= COLS )
        dir = -1;
}
```

```
/*
 * install a handler, tell kernel who to notify on input, enable
 * signals
 */
void enable_kbd_signals()
{
    int fd_flags;

    fcntl(0, F_SETOWN, getpid());
    fd_flags = fcntl(0, F_GETFL);
    fcntl(0, F_SETFL, (fd_flags|O_ASYNC));
}
```

7.12.3　方法 2：使用 aio_read

相比设置文件描述符的 O_ASYNC 位，使用 aio_read 更加灵活，当然也复杂些。对原来的弹球程序做 4 处改动。

第一，设置输入被读入时所调用的处理函数 on_input。

第二，设置 struct kbcbuf 中的变量来指明等待什么类型的输入，当输入发生时产生什么信号。在这个简单的程序中，需要从文件描述符 0 中读入一个字符，当字符被读入时希望收到 SIGIO 信号。实际上能指定任何信号，甚至是 SIGARLM 或 SIGINT。

第三，通过将以上定义的结构体传给 aio_read 来递交读入请求。和调用一般的 read 不同，aio_read 不会阻塞进程。相反 aio_read 会在完成时发送信号。

现在这个程序可以做任何它想做的事情了。在下面这个简单的例子中，只是调用 pause 来等待信号。当用户输入字符，aio_read 向进程发送 SIGIO 信号，响应处理函数被调用。

最后，实现处理函数，函数通过调用 aio_return 来得到输入的字符。然后处理这个字符。

```
/* bounce_aio.c
 * purpose animation with user control, using aio_read() etc
 * note set_ticker() sends SIGALRM, handler does animation
 * keyboard sends SIGIO, main only calls pause()
 * compile cc bounce_aio.c set_ticker.c -lrt -l curses -o bounce_aio
 */
#include  <stdio.h>
#include  <curses.h>
#include  <signal.h>
#include  <aio.h>

/* The state of the game */

#define    MESSAGE     "hello"
#define    BLANK       "     "

int row    = 10;                    /* current row */
```

```
int col    = 0;                       /* current column */
int dir    = 1;                       /* where we are going */
int delay  = 200;                     /* how long to wait */
int done   = 0;

struct aiocb kbcbuf;                  /* an aio control buf */

main()
{
   void  on_alarm(int);               /* handler for alarm */
   void  on_input(int);               /* handler for keybd */
   void  setup_aio_buffer();

   initscr();                         /* set up screen */
   crmode();
   noecho();
   clear();

   signal(SIGIO, on_input);           /* install a handler */
   setup_aio_buffer();                /* initialize aio ctrl buff */
   aio_read(&kbcbuf);                 /* place a read request */

   signal(SIGALRM, on_alarm);         /*install alarm handler */
   set_ticker(delay);                 /* start ticking

   mvaddstr( row, col, MESSAGE );     /* draw initial image */

   while( ! done )                    /* the main loop */
      pause();
   endwin();
}
/*
 * handler called when aio_read() has stuff to read
 * First check for any error codes, and if ok, then get the return code
 */
void on_input()
{
   int c;
   char *cp = (char *) kbcbuf.aio_buf; /* cast to char * */

   /* check for errors */
   if ( aio_error(&kbcbuf) != 0 )
      perror("reading failed");
   else
      /* get number of chars read */
      if ( aio_return(&kbcbuf) == 1 )
      {
```

```
            c =  *cp;
            if ( c == 'Q' || c == EOF )
                done = 1;
            else if ( c == ' ' )
                dir = -dir;
        }

    /* place a new request */
    aio_read(&kbcbuf);
}

void on_alarm()
{
    signal(SIGALRM, on_alarm);              /* reset, just in case */
    mvaddstr( row, col, BLANK );            /* clear old string */
    col += dir;                             /* move to new column */
    mvaddstr( row, col, MESSAGE );          /* draw new string */
    refresh();                              /* and show it */
    /*
     * now handle borders
     */
    if ( dir == -1 && col <= 0 )
        dir = 1;
    else if ( dir == 1 && col + strlen(MESSAGE) >= COLS )
        dir = -1;
    }
    /*
     * set members of struct.
     * First specify args like those for read(fd, buf, num) and offset
     * Then specify what to do (send signal) and what signal (SIGIO)
     */
    void setup_aio_buffer()
    {
        static char input[1];                   /* 1 char of input */
        /* describe what to read */
        kbcbuf.aio_fildes     = 0;              /* standard intput */
        kbcbuf.aio_buf        = input;          /* buffer */
        kbcbuf.aio_nbytes     = 1;              /* number to read */
        kbcbuf.aio_offset     = 0;              /* offset in file */

        /* describe what to do when read is ready */
        kbcbuf.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
        kbcbuf.aio_sigevent.sigev_signo = SIGIO; /* send sIGIO */
}
```

7.12.4 弹球程序中需要异步读入吗

不。用户输入阻塞程序，间隔计时器驱动球移动的模式工作的很好。异步读入的优势在于程序不用被输入阻塞而可以做些其他什么。

比如一个更有想象力的程序可以用这段时间放音乐、生成声响效果、计算复杂的背景图案，甚至于做一些公共服务。越来越多的计算机贡献出空闲时间来帮助计算数学、天文学或医学领域的一些大计算量的工作。

弹球程序可以用它的空闲时间计算任意精度的 pi 值、用异步输入来响应用户的键盘输入。对 main 中的循环做以下修改：

改动之前的程序：	改动之后的程序：
`while(! done)` `  pause();` `endwin();`	`compute_pi();` `endwin();`

修改后监督程序调用函数来计算 pi 值。当接收到一个字符，程序跳至处理函数来处理输入，然后继续计算。当接收到一个计时器间隔消息，程序跳到相应的处理函数去处理计时消息，处理结束后继续计算 pi 值。

如这个程序需要用新的方法来处理“Q”键被按下的消息。怎样修改程序来达到这个目的呢？

7.12.5 异步输入、视频游戏和操作系统

本章开始的时候对视频游戏和操作系统做了对比。本章的弹球游戏不需要异步输入，但操作系统需要。内核要运行程序而不能把时间浪费在等待用户输入上。内核设置当键盘、串口或网卡得到输入时被调用的处理函数。内核从一个运行中的程序跳转到处理函数，处理输入，再跳回运行中的程序。在临界区，内核阻塞信号。

内核的异步输入是由硬件实现的，而进程的异步输入是由软件实现的。它们之间有什么联系吗？游戏正在运行。突然用户按下一个键，一个电子信号被送到键盘端口。键盘端口产生一个真实的硬件信号。这个信号引发控制从视频游戏的运行中转到键盘的设备驱动。

内核的设备驱动代码从输入端口读入字符，然后将读入的字符通过终端驱动进行处理。如果驱动的文件描述符被设置为异步输入，内核向进程发送信号。当进程继续运行时，控制转移到进程内的信号处理函数。

小　　结

1. 主要内容

- 有些程序的控制流很简单。而另外一些则要响应外部的事件。一个视频游戏要响应时钟和用户输入。操作系统也要响应时钟和外设。

- curses 库有一些可以管理屏幕显示字符的函数。
- 一个进程通过设置计时器来安排事件。每个进程有 3 个独立的计时器。计时器通过发送信号来通知进程。每个计时器都可以被设置为只发送一次信号，或者按固定的间隙发送信号。
- 处理一个信号很简单。同时处理多个信号就复杂些了。进程能决定是忽略信号还是阻塞信号。进程能够告知内核哪些信号在什么时候阻塞或忽略。
- 有些函数执行一些复杂的任务是不能被打断的。程序可以通过小心地使用信号掩码来保护这些临界区代码。

2. 进一步的问题

本章中，了解到视频游戏是如何通过接收和处理信号来同时做几件事情的。Unix 同时运行几个程序。进程是如何运行的？进程来自何处？下面将注意力从操作系统的基本概念和原则转到如何创建进程、如何在进程间通信这些细节上。

3. 习题

7.1 pause 等待任何信号的到达，包括从键盘生成的信号，比如 SIGINT。

(1) 运行 sleep1 然后按下 Ctrl-C。会有什么现象？为什么？

(2) 修改 sleep1 以处理 SIGINT。

(3) 再运行程序，按下 Ctrl-C 又会如何？为什么？

7.2 在有些情况下，慢速设备(Slow Devices)的 read 系统调用可以被中断。比如用户可以通过按下 Ctrl-C 来中断程序从键盘读入。而另一些情况下，比如程序在调用 read 从磁盘读入，按下 Ctrl-C 则不会中断系统调用。

通过读手册或者查找 Web 来学习有关慢速设备的知识。哪些 read 的调用可以被中断？哪些不能？为什么？

7.3 sigprocmask 和 ISIG 另一种方法来防止重入临界区代码不被键盘信号打断的方法是关掉 tty 驱动中的 ISIG。这个方法和在信号掩码上添加这些信号有什么不同？

7.4 发明一种可重入的系统，该系统支持人们到你的办公室来将他们的名字和地址登记在列表中。试给出一种算法，该算法中信号处理函数在每次被调用时向一个文本文件的末尾添加 3 行数据。看看第 5 章有关文件自动增长模式(auto-append mode)和采用 link 来锁住文件的练习。

7.5 讨论如果 bounce1d.c 中执行 move_msg 一次的时间比计时器的间隔要长的情况下会如何。变量 pos 会如何？屏幕会有什么表现？在阻塞和递归两种情况下回答这些问题。有没有既不丢失信号同时又防止数据损毁的方法？

7.6 在一些版本的 Unix 中，计时器信号被处理的话会中断对 getch 的调用。这些系统中被计时器信号中断的 getch 会返回 EOF。这对程序有什么影响？这些影响会产生问题吗？能弥补吗？

7.7　异步输入版的弹球有两个信号处理函数。如果 SIGIO 在程序处理 SIGALRM 的时候到达会如何？反之又如何呢？两个处理函数会相互影响吗？在处理信号的时候要阻塞其他信号吗？递归调用又如何？如果新的字符在程序处理 SIGIO 时到达会不会有问题？

列出所有可能的组合，同时列出可能引发的问题。

4. 编程练习

7.8　有些浏览器支持闪烁字符和移动字符(theater-marquee text)。修改 hello1.c 以显示闪烁字符。如果用户在命令行提供一段字符串，程序应该显示提供的字符串，否则就显示默认的字符串。用 sleep 让程序在打印和擦除字符串之间停顿。

7.9　使用 curses 来实现移动字符效果，并用这个效果显示一个文件的内容。移动字符效果(Theater Marquee)(或者传动带效果 Ticker Tape)是在一个水平区域中水平地逐字地滚动显示字符串。程序应该能从命令行中读入文件名和显示串的长度、位置及速度。

7.10　修改 hello5.c，用 usleep 来替代 sleep，选择一个能够平滑但不太快的移动时间间隔值。

修改程序使字符串在屏幕两边移动变慢，而到中间就移动加快。看看程序产生的轨迹能多接近单摆或弹簧上的物体的简谐振动的轨迹。

想象屏幕的右边是行星，字符串从太空落下。修改程序使之能模拟重力的加速作用。可以做的更加真实些，在字符串撞击地面时碎裂成四处飞溅的小字符。

7.11　程序 ticker_demo.c 从信号处理函数退出。能否修改程序使之从 main 函数退出而不是信号处理函数。添加一个名为 done 的全局变量。然后对原来的程序再做两处修改使之能从 main 退出。两种方法的利弊各是什么？

7.12　修改 sigdemo3.c，把两个信号处理函数合并成一个。在合并后的处理函数中通过检查参数来判断是什么类型的信号触发处理函数的调用。这一改动会对程序的行为有何影响？

7.13　你有连到远端机器，然后忘了退出的经历吗？一个后台运行的程序在一定的空闲时间后向登录命令解释进程发送 SIGKILL 消息会很有用。

(1) 写程序 timeout.c。这个程序接受命令行参数包括进程 ID 和秒数。程序睡眠指定的秒数后向指定的进程发送 SIGKILL 信号。可以从命令解释进程用 timeout $$　3600 & 命令来启动程序。符号$$表示命令解释进程的 ID。

(2) timeout.c 的问题是一个小时以后就算你还在工作它还是会让你退出。修改程序使之只在你的 tty 没有输入/输出 10 分钟以后才退出。(提示：/dev/ttyxx 的修改时间代表了最后一次从设备读入或写出数据的时间。修改程序使之能够以参数的形式接受 tty 的名字。)

7.14 在本练习中将写一个程序在用户层模拟图 7.14 演示的情况。说明如何由一个实际时钟驱动两个不同的计时器。

首先，基于 sigdemo1.c 写一个名为 ouch.c 的程序。也就是第 6 章中的 OUCH 程序。ouch.c 将从命令行接受两个参数。一个是信号处理函数要打印的字符串，另一个是信号处理函数在打印之前要被调用的次数。比如命令：

```
$ ouch hello 10 &
```

将在后台运行程序，程序每收到 10 次 SIGINT 信号打印一次 hello。

然后写一个节拍程序，并命名为 metronome.c。这个程序从命令行接受一个进程 ID 列表。程序用间隔计时器来生成间隔为 1 秒的 SIGALRM 信号。处理函数利用 kill 系统调用向指明的进程发送 SIGINT 信号。比如命令：

```
$ metronome 1 3456 7777 2345
```

将每隔 1 秒钟向 ID 为 3456、7777 和 2345 的进程发送 SIGINT 信号。

在后台启动 3 个 ouch 实例的进程。给每个进程不同的消息字符串和间隙。然后记下它们的 ID。最后以 1 和那些进程的 ID 作为参数来启动 metronome 程序。

7.15 用 usleep()来阻塞程序，用处理函数来处理输入 在 bounce1d.c 中主循环在 getch 被阻塞，信号处理函数来控制动画。基于 hello5.c 的机制重写 bounce1d.c。在新的版本中用户输入和动画的角色将被调换，也就是说主循环在调用 usleep 被阻塞，程序在信号处理函数中处理用户输入。

7.16 写一个测试用户的反应速度的程序。程序等待一个随机的间隔时间然后在屏幕上打印一个一位十进制数字。用户要在尽可能短的时间内输入这个数字。程序记录用户的反应速度。程序进行 10 次这样的测试然后报告最慢、最快和平均响应时间。（提示：看看 gettimeofday 的用户手册）

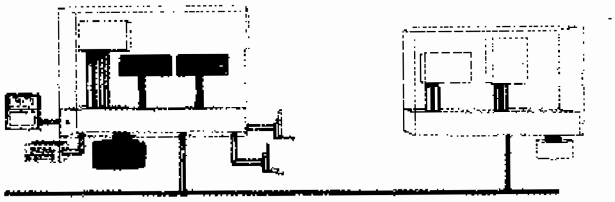
7.17 完成本章开始描述的弹球游戏。添加分数、多用户、缓冲器和其他任何能想到的使游戏变得好玩的机制。

5. 项目

基于本章学习的知识，能够实现以下的几个 Unix 程序了：

snake、worms

第 8 章　进程和程序：编写命令解释器 sh



概念与技巧

- Unix shell 的功能
- Unix 的进程模型
- 如何执行一个程序
- 如何创建一个进程
- 父进程和子进程之间如何通信

相关的系统调用

- fork
- exec
- wait
- exit

相关命令

- sh
- ps

8.1　进程＝运行中的程序

Unix 是如何运行程序的？这看起来很容易：首先登录，然后 shell 打印提示符，输入命令并按回车键。程序立即就开始运行了。当程序结束后，shell 打印一个新的提示符。但这些是如何实现的呢？什么是 shell？shell 做了些什么呢？内核又做些什么？程序是什么？运行一个程序意味着什么？

一个程序是存储在文件中的机器指令序列。一般它是由编译器将源代码编译成二进制格式的代码。运行一个程序意味着将这个机器指令序列载入内存然后让处理器(CPU)逐条执行这些指令。

在 Unix 术语中，一个可执行程序是一个机器指令及其数据的序列。一个进程是程序运行时的内存空间和设置。图 8.1 显示了程序和进程。

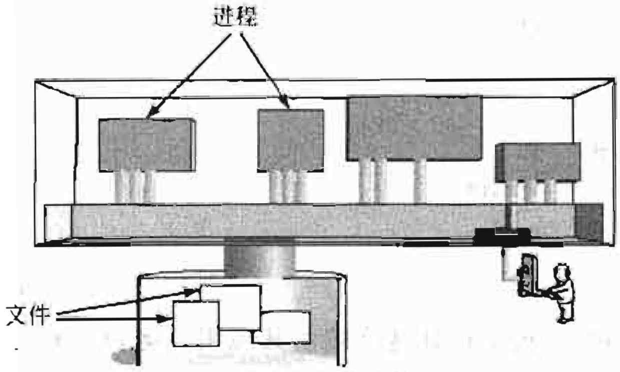


图 8.1　系统中的进程和程序

数据和程序存储在磁盘文件中，程序在进程中运行。以下的几章里将学习进程概念。从命令 ps 和 sh 开始，然后写一个自己的 Unix shell。

8.2　通过命令 ps 学习进程

进程存在于用户空间。用户空间是存放运行的程序和它们的数据的一部分内存空间。如图 8.2 所示，可以通过使用 ps(process status 进程状态的简写)命令来查看用户空间的内容。这个命令会列出当前的进程。

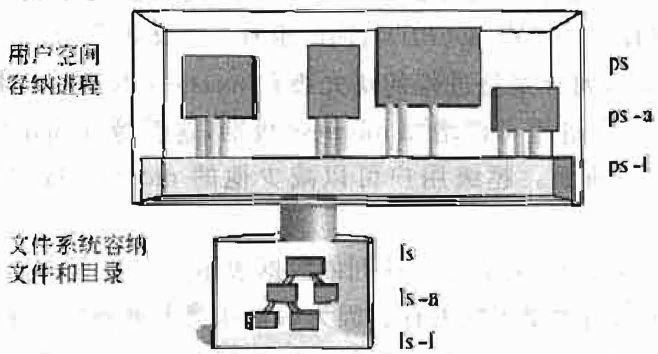


图 8.2　ps 命令列出当前进程

```
$ ps
PID     TTY     TIME      CMD
1775    pts/1   00:00:17  bash
1981    pts/1   00:00:00  ps
```

这里有两个进程在运行：bash(shell)和 ps 命令。每个进程都有一个可以惟一标识它的数字，被称为进程 ID。一般简称为 PID。每个进程都与一个终端相连，这里是/dev/pts/1。每个进程都有一个已运行的时间。注意 ps 对已运行时间统计并不是非常的精确，从 ps 只用了 0 秒就可以看出。

ps 有很多可选项。和 ls 命令一样，ps 支持 -a 可选项：

```
$ ps -a
PID   TTY     TIME      CMD
1779  pts/0   00:00:13  gv
1780  pts/0   00:00:07  gs
1781  pts/0   00:00:01  vi
2013  pts/2   00:00:23  xpaint
2017  pts/2   00:00:02  mail
2018  pts/1   00:00:00  ps
```

-a 选项列出所有进程,包括在其他终端由其他用户运行的程序。但是带选项 -a 的输出并不包括 shell。ps 也有一个 -l 选项来打印更多细节:

```
$ ps -la
F    S  UID   PID    PPID   C  PRI  NI  ADDR  SZ    WCHAN    TTY    TIME      CMD
000  S  504   1779   1731   0  69   0   -     1086  do_sel   pts/0  00:00:13  gv
000  S  504   1780   1779   0  69   0   -     2309  do_sel   pts/0  00:00:07  gs
000  S  504   1781   1731   0  72   0   -     1320  do_sel   pts/0  00:00:01  vi
000  S  519   2013   1993   0  69   19  -     1300  do_sel   pts/2  00:00:23  xpain
000  S  519   2017   1993   0  69   0   -     363   read_c   pts/2  00:00:02  mail
000  R  500   2023   1755   0  79   0   -     750   -        pts/1  00:00:00  ps
```

名为 S 的一列表示各个进程的状态。S 列的值为 R 说明 ps 对应的进程正在运行。其他进程的 S 列值都是 S,说明它们都处于睡眠状态。每个进程都属于相应的由 UID 列指明的用户 ID。每个进程都有一个进程 ID(PID),同时也有一个父进程 ID(PPID)。

标记为 PRI 和 NI 的列分别是进程的优先级和 niceness 级别。内核根据这些值来决定什么时候运行进程。一个进程可以增加 niceness 级别,这就像在超市里在排队付账的时候让其他客户排到自己的前面。超级用户可以减少他的 niceness 级别,这就像排队的时候插队。

一个进程有大小,这由 SZ 列表示。这列的数据表示这个进程占用的内存大小。在例子中 mail 程序比 xpaint 占用少得多的内存。因为后者耗费大量的内存来存储影像。程序在运行的时候占用的内存数量可能会动态的改变。如果程序在运行时分配内存,那么它占用的内存就会增加。

WCHAN 列显示进程睡眠的原因。上面的例子中所有睡眠的进程都是等待输入。read_c 或 do_sel 代表内核的地址。ADDR 和 F 已经不再用了,但是为了兼容的原因而保留它们。选项 -ly 将只显示目前使用的值。

各个版本的 Unix 间的可选命令参数差别很大。前面提到的 -a 和 -l 可选项可能在你的系统中不能用或者结果不同。查阅你的用户手册。上面的例子是来自于版本号为 procps 2.0.6。例子只给出了 ps 的一部分显示功能。

ps 命令是很强大的。-fa 可选项产生如下输出:

```
$ ps -fa
UID        PID     PPID  C  STIME TTY          TIME     CMD
```

```
betsy     1779   1731  0  19:53 pts/0    00:00:01    gv dinner.ps
betsy     1980   1779  0  19:53 pts/0    00:00:07    gs -dNOPLATFONTS
betsy     1781   1731  0  19:54 pts/0    00:00:02    vi dinner
yuriko    2013   1993  0  20:15 pts/2    00:00:00    xpaint
yuriko    2017   1993  0  20:16 pts/2    00:00:00    mail bruce
bruce     2401   1755  0  20:36 pts/1    00:00:00    ps -af
```

-f 表示格式化输出，这样便于阅读。用用户名代替 UID 来显示。在 CMD 列显示完整的命令行。

8.2.1 系统进程

除了用户运行的进程外，其他一些是 Unix 系统用来完成系统任务的进程：

```
$ ps -ax | head -25
PID TTY    STAT    TIME    COMMAND
  1 ?      S       0:05    init
  2 ?      SW      3:54    [kflushd]
  3 ?      SW      0:38    [kupdate]
  4 ?      SW      0:00    [kpiod]
  5 ?      SW      2:13    [kswapd]
 35 ?      SW      0:00    [uhci-control]
 36 ?      SW      0:00    [khubd]
420 ?      S       0:25    syslogd
423 ?      S       0:36    klogd -k /boot/System.map-2.2.14
437 ?      SW      0:00    [inetd]
449 ?      S       0:02    amd -f /etc/am.d/conf
461 ?      SW      0:00    [rpciod]
466 ?      S       0:00    cron
471 ?      S       0:00    atd
476 ?      S       0:00    sendmail: accepting connections on port 25
484 ?      SW      0:00    [rpc.rstatd]
500 ?      S       0:46    sshd
504 ?      SW      0:00    [calserver]
506 ?      SW      0:00    [keyserver]
512 ?      SW      0:00    [portsentry]
514 ?      SW      0:00    [portsentry]
561 tty1   SW      0:00    [getty]
562 tty2   SW      0:00    [getty]
563 tty3   SW      0:00    [getty]
$ ps -as|wc -l
     82
```

上面的例子显示了当前系统中运行的 82 个进程中的前 24 个。其中部分是系统进程。系统进程中的很大一部分是没有终端与之相连的。它们在系统启动时启动，而不是由用户

在命令行输入。这些系统进程做些什么呢?

列表中开始的几个分别处于内存的不同部分,包括内核缓冲和虚存页面。列表中的其他一些管理系统日志(klogd,syslogd)、调度批任务(cron,atd)、防范可能的攻击(portsentry)和让一般的用户登录(sshd,getty)。可以通过 ps -ax 的输出和 Unix 手册了解很多系统的情况。运行 ps 就像透过显微镜看一滴池塘水。能看到很多各式各样的进程运行在系统中。

8.2.2 进程管理和文件管理

从运行 ps 的结果看出进程有很多属性。每个进程属于某个用户 ID、有一定的大小、一个起始时间、已运行的时间、优先级和 niceness 级别。有些进程与某个终端相连,而其他一些则没有。这些属性存放在什么地方呢? 曾对文件提过同样的问题。内核管理内存中的进程和磁盘上的文件。这些管理活动有什么相似之处吗?

文件包含数据,进程包含可执行代码。文件有一些属性,进程也有一些属性。内核建立和销毁文件,进程类似。就像管理磁盘的多个文件,内核管理内存中的多个进程,为它们分配空间,并记录内存分配情况。内存管理和磁盘管理有什么相似之处?

8.2.3 内存和程序

进程这个概念有些抽象,但是它代表了一些非常实际的实体:内存中的一些字节。图 8.3 演示了计算机内存的 3 种模式。

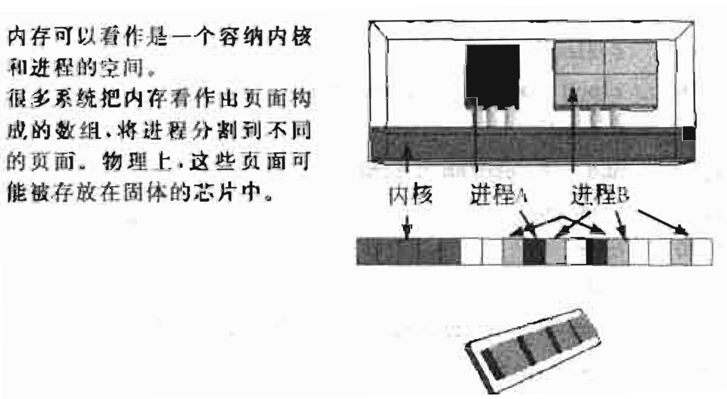


图 8.3　计算机内存的 3 种模式

Unix 系统中的内存分为系统空间和用户空间。进程存在于用户空间。内存实际上就是一个字节序列,或者一个很大的数组。如果机器有 64 MB 的内存,那意味着这个数组有大约 6700 万个内存位置。其中的一些用来存放组成内核的机器指令和数据。

还有一些存放组成进程的机器指令和数据。一个进程不一定必须要占一段连续的内存。就像文件在磁盘上被分成小块,进程在内存也被分成小块。同样和文件有记录分配了的磁盘块的列表相似,进程也有保存分配到的内存页面(memory pages)的数据结构。因此,将进程表示为用户空间内的一个小方块只是某种程度的抽象。

将内存表示为连续的字节数组也是一种抽象。现在的内存一般情况下是由小电路板上

的一些芯片组成。

建立一个进程有点像建立一个磁盘文件。内核要找到一些用来存放程序指令和数据的空闲内存页。内核还要建立数据结构来存放相应的内存分配情况和进程属性。

使操作系统变得神奇的不仅是它的文件系统把一堆旋转圆盘上连续的簇变成有序组织的树状目录结构，而且以相似的机制，它的进程系统将硅片上的一些位组织成一个进程社会——成长、相互影响、合作、出生、工作和死亡。这有点像蚂蚁农庄。

为了理解进程，下面将学习和实现一个 Unix shell。shell 是一个管理和运行程序的程序。

8.3　shell：进程控制和程序控制的一个工具

shell 是一个管理进程和运行程序的程序。就像存在很多编程语言，Unix 系统有很多种可用的 shell，每种都有各自的风格和优势。所有常用的 shell 都有三个主要功能：

（1）运行程序

（2）管理输入和输出

（3）可编程

看看下面的命令序列：

```
$ grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
$ TZ = PST8PDT;export TZ;date;TZ = EST5EDT
Sat Jul 28 02:10:05 PDT 2001
$ date
Sat Jul 28 05:10:14 EDT 2001
$ ls -l /etc > etc.listing
$ NAME = lp
$ if grep $ NAME /etc/passwd
>then
> echo hello | mail $ NAME
>fi
lp:x:4:7:lp:/var/spool/lpd:
$
```

（1）运行程序

grep、date、ls、echo 和 mail 都是一些普通的程序，用 C 编写，并被编译成机器语言。shell 将它们载入内存并运行它们。很多人把 shell 看成是一个程序启动器(program launcher)。

（2）管理输入和输出

shell 不仅仅是运行程序。使用<、>和 | 符号可以将输入、输出重定向。这样就可以告诉 shell 将进程的输入和输出连接到一个文件或是其他的进程。

（3）编程

shell 同时也是带有变量和流程控制的编程语言。在上面的例子中，可以看到使用了两个变量。首先，变量 TZ 被设置成表示美国西海岸时区的字符串。然后这个值被作为参数传给 date 命令来打印当前的日期和时间。

例子的后面部分，可以看到有 if.. then 语句。变量 NAME 被置为字符串“lp”。$ NAME 的值在 grep 命令中被使用。grep 的结果由 if 语句进行判断。如果在文件/etc/passwd 中搜索到字符串“lp”，shell 就执行命令 echo hello| mail $ NAME。否则跳至下一条命令。

在本章中，先来看看 shell 是如何运行一个程序的。在后面的章节中将学习 shell 的脚本语言和输入、输出的重定向。

8.4　shell 是如何运行程序的

shell 打印提示符，输入命令，shell 就运行这个命令，然后 shell 再次打印提示符——如此反复。那么这些现象的背后到底发生些什么?

一个 shell 的主循环执行下面的 4 步(如图 8.4 所示)：

(1) 用户键入 a.out;

(2) shell 建立一个新的进程来运行这个程序;

(3) shell 将程序从磁盘载入;

(4) 程序在它的进程中运行直到结束。

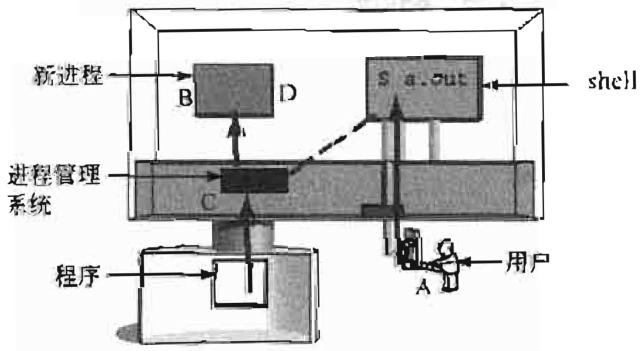


图 8.4　用户要求 shell 运行一个程序

8.4.1　shell 的主循环

shell 由下面的循环组成：

```
while( ! end_of_input)
    get command
    execute command
    wait for command to finish
```

考虑下面这个与 shell 典型的互动：

```
$ ls
Chap.bak Story08.tr chap08.ps      chap08.tr outline.08
Makefile chap08      chap08.short  code       pix
$ ps
PID      TTY      TIME       CMD
29182    pts/5    00:00:00  bash
29183    pts/5    00:00:00  ps
$
```

用图 8.5 的时间轴来表示事件发生次序。其中时间从左向右消逝。shell 由标识为 sh 的方块代表，它随着时间的流逝从左到右移动。shell 从用户读入字符串“ls”。shell 建立一个新的进程，然后在那个进程中运行 ls 程序并等待那个进程结束。

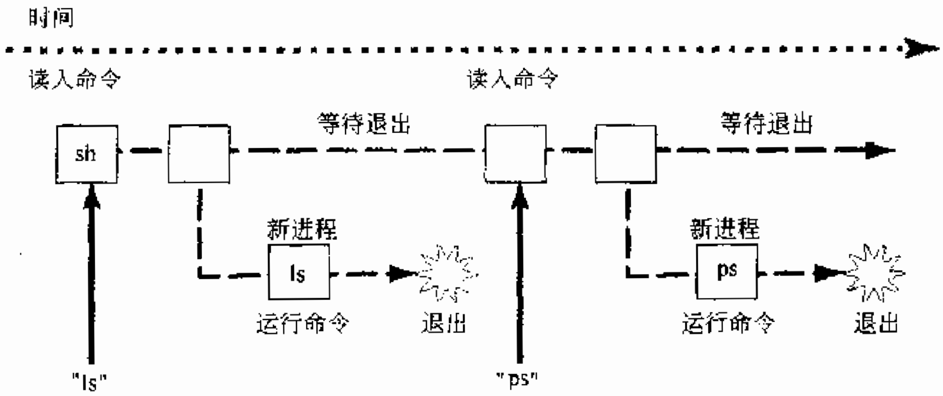


图 8.5　shell 主循环的时间轴

然后 shell 读入新的一行输入，建立一个新进程，在这个进程中运行程序并等待这个进程结束。

当 shell 检测到输入结束，它就退出。

为了要写一个 shell，需要学会：

(1) 运行一个程序；

(2) 建立一个进程；

(3) 等待 exit()。

当学会这些，就能用这些技术来实现自己的 shell 了。

8.4.2　问题 1：一个程序如何运行另一个程序

答案：程序调用 execvp。

图 8.6 显示了一个程序如何运行另一个程序。比如，为了运行 ls -la，一个程序调用 execvp("ls",arglist)。这里 arglist 是命令行的字符串数组。内核从磁盘将程序载入内存。命令参数 ls 和 -la 被传给程序，然后程序开始运行。简而言之：

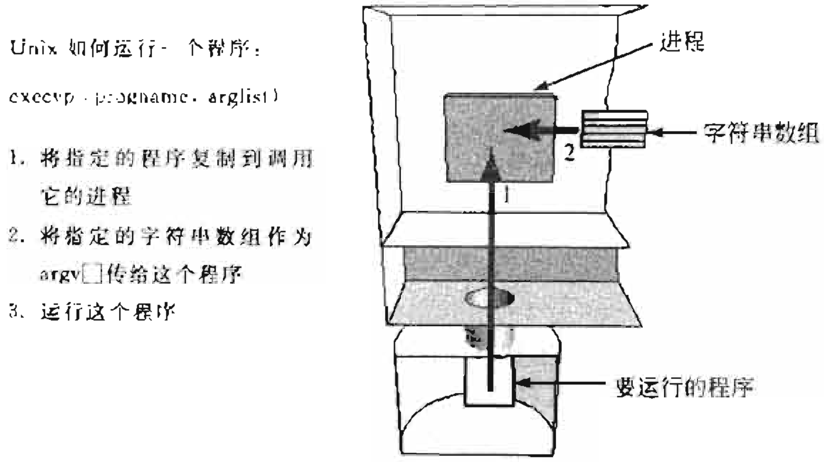


图 8.6　execvp 将程序复制到内存后运行它

(1) 程序调用 execvp

(2) 内核从磁盘将程序载入

(3) 内核将 arglist 复制到进程

(4) 内核调用 main(argc,argv)

下面是运行 ls -l 的完整程序：

```
/* execl.c - shows how easy it is for a program to run a program
 */

main()
{
    char * arglist[3];

    arglist[0] = "ls";
    arglist[1] = "-l";
    arglist[2] = 0 ;
    printf("* * * About to exec ls -l\n");
    execvp( "ls" , arglist );
    printf("* * * ls is done. bye\n");
}
```

execvp 有两个参数：要运行的程序名和那个程序的命令行参数数组。当程序运行时命令行参数以 argv[]传给程序。注意，将数组的第一个元素置为程序的名称。还要注意，最后一个元素必须是 null。

编译并运行这个程序：

```
$ cc execl.c -o execl
$ ./execl.c
*** About to exec ls -l
```

```
total 28
drwxr-x---      2 bruce     users      1024 Jul 14 21:02 a
drwxr-x---      3 bruce     users      1024 Jul 16 03:16 c
-rw-r--r--      1 bruce     users         0 Jul 14 21:03 y
$
```

1. 第二条打印的消息哪里去了？

再看一下代码。程序宣布它要运行 ls 程序，运行 ls 程序，然后宣布 ls 运行结束。那么第二条信息呢？

一个程序在一个进程中运行——也就是一些内存和内核中相应的数据结构。这样，execvp 将程序从磁盘载入进程以便它可以被运行。但是载入到哪个进程呢？这就是问题之所在：内核将新程序载入到当前进程，替代当前进程的代码和数据。

2. execvp 就像换脑

有人可能会有这样的愿望："我希望能用爱因斯坦的脑子解决这个问题，然后再用自己的脑子做其他的事情。"一种实现这个愿望的方法是拿掉你的大脑，然后装上爱因斯坦的大脑。这样你就拥有了爱因斯坦的思想和分析能力。这样想拥有两个思维的愿望就和原来的大脑[①]一起被拿掉了。

exec 系统调用[②]从当前进程中把当前程序的机器指令清除，然后在空的进程中载入调用时指定的程序代码，最后运行这个新的程序。exec 调整进程的内存分配使之适应新的程序对内存的要求。相同的进程，不同的内容。

execvp()总结如下。

execvp		
目标	在指定路径中查找并执行一个文件	
头文件	#include <unistd.h>	
函数原型	result = execvp(const char * file,const char * argv[])	
参数	file	要执行的文件名
	argv	字符串数组
返回值	−1	如果出错

execvp 载入由 file 指定的程序到当前进程，然后试图运行它。execvp 将以 NULL 结尾的字符串列表传给程序。execvp 在环境变量 PATH 所指定的路径中查找 file 文件。

如果执行成功，execvp 没有返回值。当前程序从进程中清除，新的程序在当前进程中运行。

3. 带提示符的 shell

前面所学已经足够写第一个版本的 shell 了。这里已经知道如何运行一个程序，还知道如何将命令行参数传给它。第一个 shell 提示用户输入程序名和参数，然后运行指定的程序。

① 和所有原有大脑记忆中要做的事情。

② execvp 是一组基于 execve 系统调用函数中的一个，它们统称为 exec。

将程序命名为 psh1. c，它是带提示符的 shell(prompting shell)的缩写：

```
/* prompting shell version 1
 * Prompts for the command and its arguments.
 * Builds the argument vector for the call to execvp.
 * Uses execvp(), and never returns.
 */

#include  <stdio.h>
#include  <signal.h>
#include  <string.h>

#define     MAXARGS      20         /* cmdline args*/
#define     ARGLEN       100        /* token length*/

int main()
{
   char     *arglist[MAXARGS+1];        /* an array of ptrs*/
   int      numargs;                    /* index into array*/
   char     argbuf[ARGLEN];             /* read stuff here*/
   char     *makestring();              /* malloc etc */
   numargs = 0;
   while ( numargs < MAXARGS )
   {
      printf("Arg[%d]? ", numargs);
      if ( fgets(argbuf, ARGLEN, stdin) && *argbuf != '\n' )
         arglist[numargs++] = makestring(argbuf);
      else
      {
         if ( numargs > 0 ){             /* any args? */
            arglist[numargs]=NULL;       /* close list */
            execute( arglist );          /* do it */
            numargs = 0;                 /* and reset */
         }
      }
   }
   return 0;
}

int execute( char *arglist[] )
/*
 * use execvp to do it
 */
{
   execvp(arglist[0], arglist);         /* do it */
   perror("execvp failed");
```

```
    exit(1);
}

char * makestring( char * buf )
/*
 * trim off newline and create storage for the string
 */
{
    char * cp, * malloc();

    buf[strlen(buf) - 1] = '\0';        /* trim newline */
    cp = malloc( strlen(buf) + 1 );     /* get memory */
    if ( cp == NULL ){                  /* or die */
        fprintf(stderr,"no memory\n");
        exit(1);
    }
    strcpy(cp, buf);                    /* copy chars */
    return cp;                          /* return ptr */
}
```

psh1.c 是 Unix shell 的第一个草案。psh1 要求每个字符串单独的输入，第一个是程序名，然后依次是程序参数。代码包括两步：(1) 一个字符串一个字符串的构造参数列表 arglist，最后在数组末尾加上 NULL；(2)将 arglist[0]和 arglist 数组传给 execvp，如图 8.7 所示。

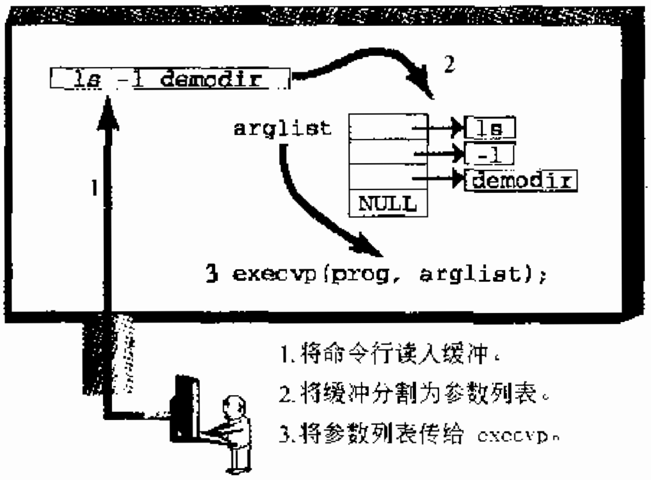


图 8.7　用数组构造 arglist

编译并运行程序：

```
$ cc psh1.c -o psh1
$ ./psh1
Arg[0]? ls
Arg[1]? -l
```

```
Arg[2]? demodir
Arg[3]?
total 2
drwxr-x---     2 bruce    users        1024 Jul 14 21:02 a
drwxr-x---     3 bruce    users        1024 Jul 16 03:16 c
-rw-r--r--     1 bruce    users           0 Jul 14 21:03 y
$
```

4. 它怎么退出了？

程序运行正常，但是就像设想的那样，execvp 用命令指定的程序代码覆盖了 shell 的程序代码，然后在命令指定的程序结束之后退出。这样 shell 就不能再次接受新的命令。为了运行新的命令，用户不得不再次运行 shell。

shell 如何能做到在运行程序的同时还能等待下一个命令呢？方法之一就是启动一个新的进程，由这个进程来执行命令程序。

8.4.3 问题 2：如何建立新的进程

答案：一个进程调用 fork 来复制自己。

用法：fork();　　/*takes no arguments*/

1. 解释 fork

继续用爱因斯坦大脑思考的问题做比喻。就像先前看到的，将爱因斯坦的大脑放到你的脑壳里不但把爱因斯坦的思想给了你，同时也清除了所有你原来大脑里的思想。

解决的方法之一就是复制一个自己，采用三位影像复制技术，逐个原子的复制一个完全等价的自己。当你建立了自己的复制品，将爱因斯坦的大脑放到它的脑壳里，这样你就可以继续你原来的计划和思考了。在你生命中的某个阶段，世界上只有一个你。当你按下复制机的复制按钮，世界上有了两个你。对自己的复制有点像马路上的岔口。开始只有一条路，然后有了两条。

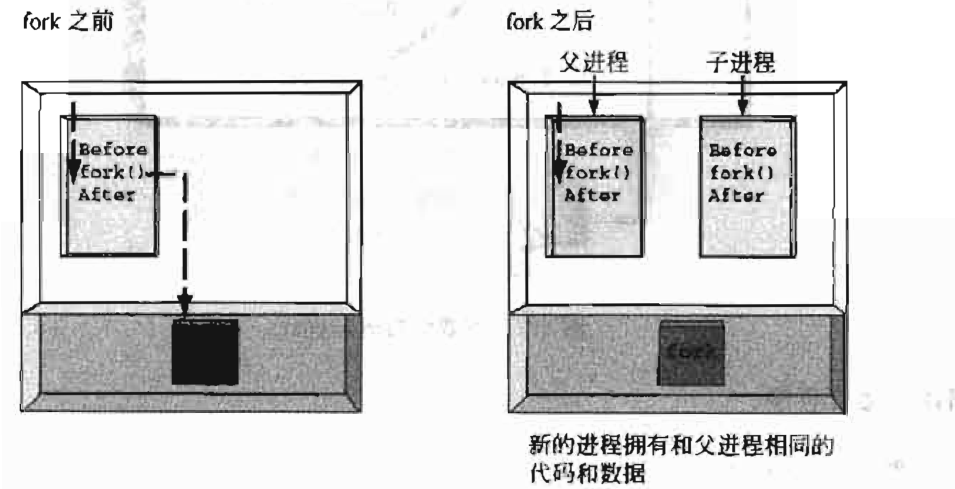


图 8.8　fork()复制一个进程

这就是系统调用 fork 做的事情。图 8.8 显示了 fork 调用前后的系统状况。进程拥有程序和当前运行到的位置。进程调用 fork，当控制转移到内核中的 fork 代码后，内核做：

(1) 分配新的内存块和内核数据结构

(2) 复制原来的进程到新的进程

(3) 向运行进程集添加新的进程

(4) 将控制返回给两个进程

当你按下复制机的开始按钮后，世界上将有两个你，物理上、心理上都是相同的。但是每个人将开始你(他)自己的人生之路。类似地，当一个进程调用 fork 之后，就有两个二进制代码相同的进程。而且它们都运行到相同的地方。但是每个进程都将可以开始它们自己的旅程。看如下程序。

2. 例子：forkdemo1.c——建立一个新的进程

forkdemo1.c 有两句打印语句，一句在 fork 之前，一句在 fork 之后：

```
/* forkdemo1.c
 * shows how fork creates two processes, distinguishable
 * by the different return values from fork()
 */

#include   <stdio.h>

main()
{
   int ret_from_fork, mypid;

   mypid = getpid();                              /* who am i? */
   printf("Before: my pid is %d\n", mypid);       /* tell the world */

   ret_from_fork = fork();

   sleep(1);
   printf ("After: my pid is %d, fork() said %d\n",
         getpid(), ret_from_fork);
}
```

如果这是一个普通的程序，将看到两行输出，每行打印一个状态。但是当它运行时将看到：

```
$ cc forkdemo1.c -o forkdemo1
$ ./forkdemo1
Before:my pid is 4170
After:my pid is 4170,fork() said 4171
$ After: my pid is 4171,fork() said 0
```

这里可以看到三行输出，一行 Before：信息和两行 After：信息。进程 4170 先是打印 Before：消息，然后它又打印 After：消息——一切正常。另一个 After：信息是由进程 4171

打印的。注意进程 4171 没有打印 Before：信息。为什么呢？用户空间的近照（如图 8.9 所示）显示了进程 4170 调用 fork 前后发生了什么。

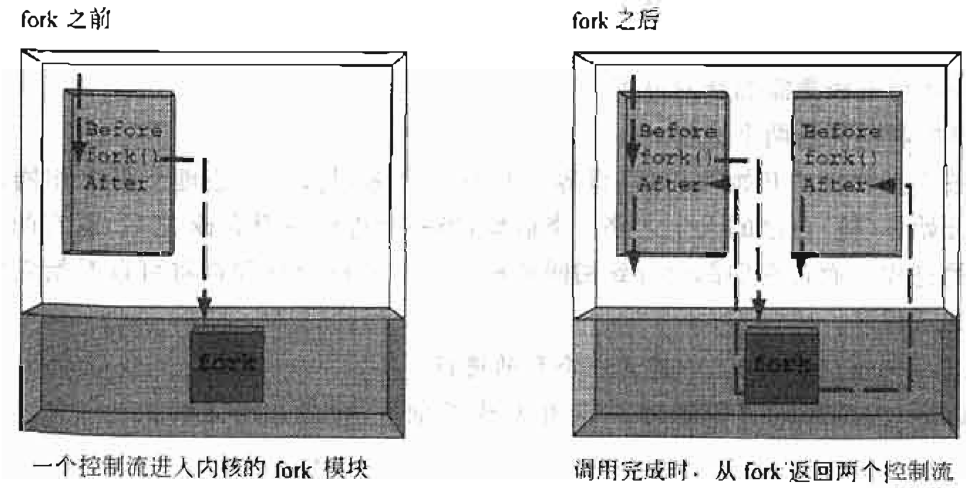


图 8.9　子进程执行 fork()之后的代码

内核通过复制进程 4170 来创建进程 4171，它将 4170 的代码和当前运行到的位置都复制给 4171。其中当前运行的位置是由随着代码向下移动的箭头表示的。新的进程 4171 从 fork 返回的地方开始运行，而不是从开头开始运行。因为 4171 是从中间开始运行的，也就不打印 Before：信息了。

3. 例子：forkdemo2. c——子进程创建进程

子进程不是从 main 函数的开始，而是从 fork 返回的地方开始它的生命之旅。预测一下下面程序会有几行输出：

```
/* forkdemo2.c - shows how child processes pick up at the return
 *              from fork() and can execute any code they like,
 *              even fork(). Predict number of lines of output.
 */

main()
{
    printf("my pid is %d\n", getpid() );
    fork();
    fork();
    fork();
    printf("my pid is %d\n", getpid() );
}
```

编译并运行这个程序，结果如何？

4. 例子：forkdemo3. c——分辨父进程和子进程

从 forkdemo1. c 可以看到进程 4170 调用 fork 创立子进程，子进程 PID 是 4171。两个进程有相同的代码，运行到同一行有相同的数据和进程属性。那么如何才能分辨到底是父进

程还是子进程呢？

这两个进程不是完全相同的。从 forkdemo1.c 的输出可以看出，不同的进程，fork 的返回值是不同的。在子进程中 fork 返回 0，在父进程中 fork 返回 4171。根据 fork 的返回值进程可以很容易的判断自己是子进程还是父进程。

在下一个例子 forkdemo3.c 中，进程根据 fork 返回值的不同打印不同的消息：

```
/* forkdemo3.c - shows how the return value from fork()
 *               allows a process to determine whether
 *               it is a child or process
 */

#include <stdio.h>

main()
{
   int fork_rv;

   printf("Before: my pid is %d\n", getpid());

   fork_rv = fork();          /* create new process */

   if ( fork_rv == -1 )       /* check for error */
      perror("fork");

   else if ( fork_rv == 0 )
      printf("I am the child. my pid = %d\n", getpid());
   else
      printf("I am the parent. my child is %d\n", fork_rv);
}
```

下面是运行结果：

```
$ ./forkdemo3
Before: my pid is 5931
I am the parent. my child is 5932
I am the child. my pid = 5932
$
```

fork 小结如下。

系统调用 fork 正是解决 shell 只能运行一条命令这个问题所需要的。使用 fork，不但能够创建新的进程，而且能够分辨原来的进程和新创建的进程。新的进程能调用 execvp 来执行任何用户指明的程序。

这里明确建立一个 shell 所需三项技术中的两项。知道了如何建立进程（fork）和如何运行一个程序（execvp）。最后需要知道如何让父进程等待子进程结束。

fork	
目标	创建进程
头文件	#include<unistd.h>
函数原型	pid_t result = fork(void)
参数	没有
返回值	−1　如果错误 0　返回到子进程
	pid 将子进程的进程 ID 传给父进程

8.4.4 问题 3：父进程如何等待子进程的退出

答案：进程调用 wait 等待子进程结束。

用法：pid = wait(&status);

1. 解释 wait()

系统调用 wait 做两件事。首先，wait 暂停调用它的进程直到子进程结束。然后，wait 取得子进程结束时传给 exit 的值。

图 8.10 显示了 wait 是如何工作的。注意时间轴是自左向右的，父进程在左边开始调用 fork。内核构造子进程，这里子进程由另外一个小方块代表。子进程开始和父进程并行运行，父进程调用 wait，内核挂起父进程直到子进程结束。父进程标识为 wait 的一段暂停运行。

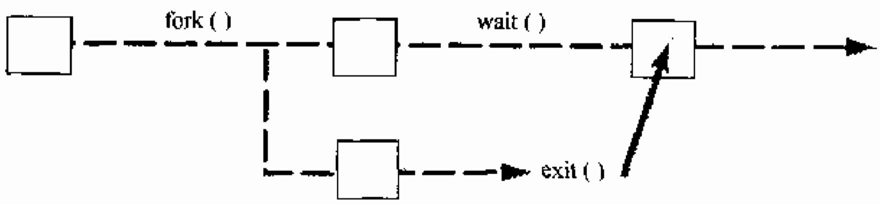


图 8.10　wait 暂停父进程直到子进程结束

最终子进程会结束任务并调用 exit(n)。n 是 0 到 255 的一个数字。

当子进程调用 exit，内核唤醒父进程同时将子进程传给 exit 的参数。唤醒和传递退出(exit)值的动作由从 exit 的括号到父进程的箭头表示。这样 wait 执行两个操作：通知和通信。

2. 例子：waitdemo1.c——通知

waitdemo1.c 显示了子进程调用 exit 是如何触发 wait 返回父进程的。

```
/* waitdemo1.c - shows how parent pauses until child finishes
 */

#include  <stdio.h>

#define  DELAY 2
```

```
main()
{
    int newpid;
    void child_code(), parent_code();

    printf("before: mypid is %d\n", getpid());

    if ( (newpid = fork()) == -1 )
        perror("fork");
    else if ( newpid == 0 )
        child_code(DELAY);
    else
        parent_code(newpid);
}
/*
 * new process takes a nap and then exits
 */
void child_code(int delay)
{
    printf("child %d here. will sleep for %d seconds\n", getpid(), delay);
            sleep(delay);
    printf("child done. about to exit\n");
    exit(17);
}
/*
 * parent waits for child then prints a message
 */
void parent_code(int childpid)
{
    int wait_rv;            /* return value from wait() */
    wait_rv = wait(NULL);
    printf("done waiting for %d. Wait returned: %d\n", childpid, wait_rv);
}
```

运行 waitdemo1.c 的结果如下：

```
$ ./waitdemo1
before: mypid is 10328
child 10329 here.will sleep for 2 seconds
child done. about to exit
do waiting for 10329. Wait returned:10329
```

运行程序、调整时间，可以发现父进程总会等到子进程调用 exit。图 8.11 演示了控制流和两个进程之间的数据传输。在父进程，控制流始于程序的开始，在 wait 的地方阻塞。在子进程，控制流始于 main 函数的中部，然后运行 child_code 函数，最后调用 exit 结束。子进程

调用 exit 就像发送一个信号给父进程以唤醒它。

父进程在 wait 处阻塞，然后在子进程退出后继续运行。

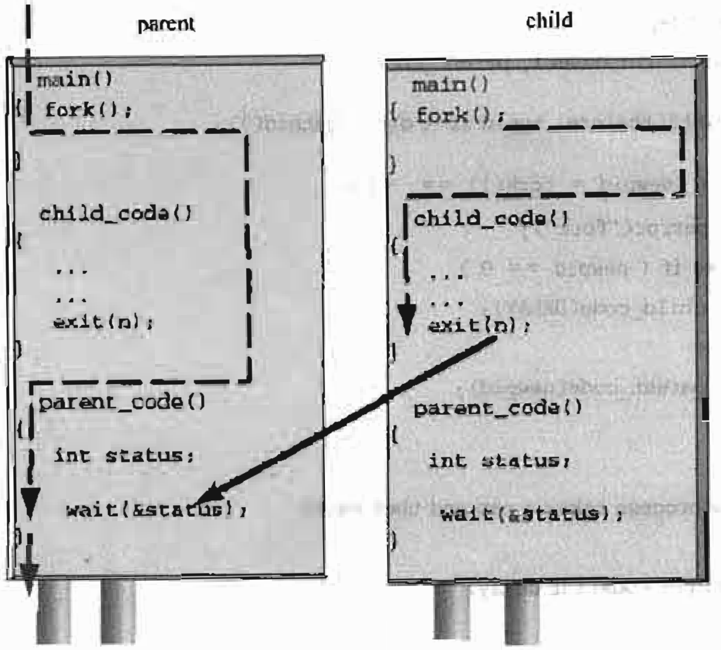


图 8.11　调用 wait()前后的控制流和进程间通信

waitdemo1.c 程序体现了 wait 的两个重要特征：

(1) wait 阻塞调用它的程序直到子进程结束

在这个简单的程序中，父进程阻塞直到子进程调用 exit。这一特征使两个进程能够同步它们的行为。比如，父进程用 fork 创建一个子进程来对一个文件排序。父进程必须等排序结束后才能继续处理这个文件。系统调用 exit 和 wait 是一种协调这些任务的方法。

(2) wait 返回结束进程的 PID

在这个简单的程序中，wait 的返回值是调用 exit 的子进程的 PID。就像在 forkdemo2.c 中看到的，一个进程可以创建多个子进程。考虑一个从两个不同的远程数据库整合数据的程序。这个程序可以使用 fork 来创建两个进程，一个用来连接并从数据库中提取数据，另一个从其他数据库提取数据。从第一个数据库提取来的数据需要做些后期处理，而从第二个数据库提取来的数据则不需要这样的处理。

wait 的返回值告诉父进程那个任务结束了。这样它就可以继续有效的处理了。

3. 例子：waitdemo2.c——通信

wait 的目的之一是通知父进程子进程结束运行了。它的第二个目的是告诉父进程子进程是如何结束的。

一个进程以 3 种方式(成功、失败或死亡)之一结束。其一，一个进程可能顺利完成它的任务。按照 Unix 惯例，成功的程序调用 exit(0)或者从 main 函数中 return 0。

其二，进程可能失败。比如进程可能由于内存耗尽而提前退出程序。按 Unix 惯例，程序遇到问题而要退出调用 exit 时传给它一个非零的值。程序员可以对不同的错误分配不同的值，手册中有详细的描述。

最后，程序可能被一个信号杀死（见第 6 和第 7 章）。信号可能来自键盘、间隔计时器、内核或者其他进程。通常的情况下，一个既没有被忽略又没有被捕获信号会杀死进程的。

wait 返回结束的子进程的 PID 给父进程。父进程如何知道子进程是以何种方式退出的呢？

答案在传给 wait 的参数之中。父进程调用 wait 时传一个整型变量地址给函数。内核将子进程的退出状态保存在这个变量中。如果子进程调用 exit 退出，那么内核把 exit 的返回值存放到这个整数变量中；如果进程是被杀死的，那么内核将信号序号存放在这个变量中。这个整数由 3 部分组成——8 个 bit 是记录退出值，7 个 bit 是记录信号序号，另一个 bit 用来指明发生错误并产生了内核映像（core dump）。图 8.12 演示了子进程状态值的 3 个部分。

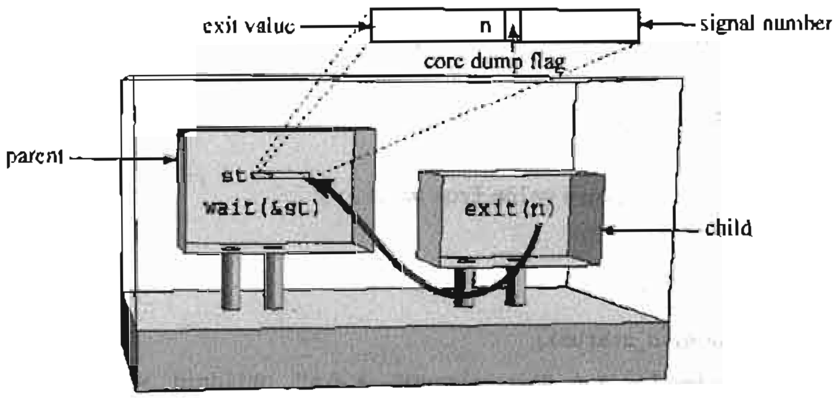


图 8.12　子进程状态值有 3 部分

例子 waitdemo2.c 是基于 waitdemo1.c 的，它显示了子进程的退出状态：

```
/* waitdemo2.c - shows how parent gets child status
 */

#include  <stdio.h>

#define   DELAY 5

main()
{
    int newpid;
    void child_code(), parent_code();

    printf("before: mypid is %d\n", getpid());

    if ( (newpid = fork()) == -1 )
        perror("fork");
    else if ( newpid == 0 )
        child_code(DELAY);
    else
```

```
        parent_code(newpid);
    }
    /*
     * new process takes a nap and then exits
     */
    void child_code(int delay)
    {
        printf("child %d here. will sleep for %d seconds\n", getpid(), delay);
        sleep(delay);
        printf("child done. about to exit\n");
        exit(17);
    }
    /*
     * parent waits for child then prints a message
     */
    void parent_code(int childpid)
    {
        int wait_rv;          /* return value from wait() */
        int child_status;
        int high_8, low_7, bit_7;

        wait_rv = wait(&child_status);
        printf("done waiting for %d. Wait returned: %d\n", childpid, wait_rv);

        high_8 = child_status >> 8;        /* 1111 1111 0000 0000 */
        low_7 = child_status & 0x7F;       /* 0000 0000 0111 1111 */
        bit_7 = child_status & 0x80;       /* 0000 0000 1000 0000 */
        printf("status: exit=%d, sig=%d, core=%d\n", high_8, low_7, bit_7);
    }
```

首先，让 waitdemo2 正常退出。退出状态从子进程处复制：

```
$ ./waitdemo2
before: mypid is 10855
child 10856 here. will sleep for 5 seconds
child done. about to exit
done waiting for 10856 . Wait returned: 10856
status: exit=17, sig=0, core=0
```

然后，在后台运行 waitdemo2，使用 kill(见第 7 章)来向子进程发送 SIGTERM 信号：

```
$ ./waitdemo2 &
$ before: mypid is 10857
child 10858 here. will sleep for 5 seconds
kill 10858
```

```
$ done waiting for 10858. Wait returned: 10858
status: exit = 0, sig = 15, core = 0
```

wait()小结如下。

wait	
目标	等待进程结束
头文件	#include <sys/types.h> #include <sys/wait.h>
函数原型	pid_t result = wait(int * statusptr)
参数	statusptr 子进程的运行结果
返回值	−1　遇到错误 pid　结束进程的进程 id
相关内容	waitpid(2), wait3(2)

wait 系统函数挂起调用它的进程直到得到这个进程的子进程的一个结束状态。结束状态是退出值或者是信号序号。如果有一个子进程已经退出或被杀死，对 wait 的调用立即返回。wait 返回结束进程的 PID。如果 statusptr 不是 NULL，wait 将退出状态或者信号序号复制到 statusptr 指向的整数中。这个值可以用<sys/wait.h>中的宏来检测。

如果调用的进程没有子进程也没有得到终止状态值，则 wait 返回−1。

8.4.5　小结：shell 如何运行程序

这一节以问题"shell 是如何运行程序的?"开始，现在已经知道答案了：shell 用 fork 建立新进程，用 exec 在新进程中运行用户指定的程序，最后 shell 用 wait 等待新进程结束。wait 系统调用同时从内核取得退出状态或者信号序号以告知子进程是如何结束的。

每个 Unix shell 都是使用图 8.13 所示的模型。现在将这 3 个系统调用组合在一起实现一个真正的 shell。

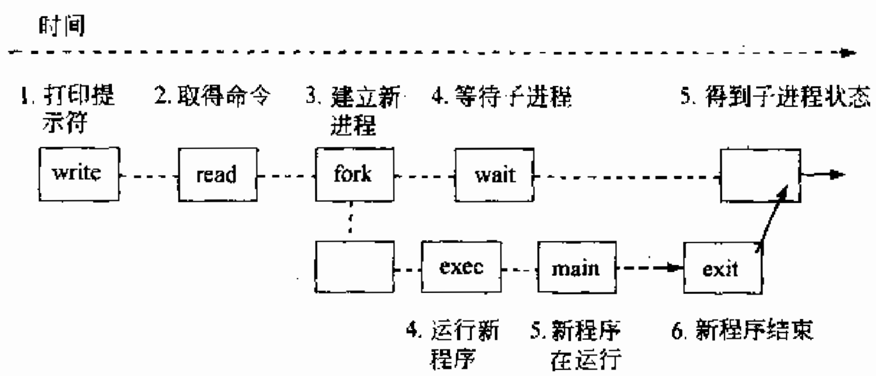


图 8.13　shell 的 fork()、exec()和 wait()循环

8.5　实现一个 shell：psh2.c

图 8.14 是一个 Unix shell 的简化流程图。下一个 shell psh2.c 将使用这个流程。

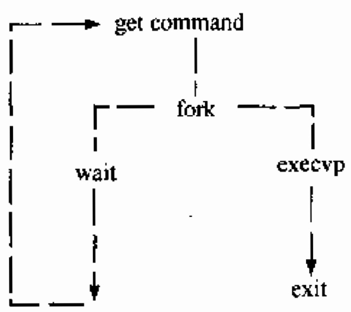


图 8.14　Unix shell 的基本逻辑

```
/** prompting shell version 2
 **
 ** Solves the 'one-shot' problem of version 1
 **     Uses execvp(), but fork()s first so that the
 **     shell waits around to perform another command
 ** New problem: shell catches signals. Run vi, press ^c.
 **/

#include  <stdio.h>
#include  <signal.h>

#define   MAXARGS     20              /* cmdline args */
#define   ARGLEN      100             /* token length */

main()
{
    char    *arglist[MAXARGS+1];      /* an array of ptrs */
    int     numargs;                  /* index into array */
    char    argbuf[ARGLEN];           /* read stuff here */
    char    *makestring();            /* malloc etc */

    numargs = 0;
    while ( numargs < MAXARGS )
    {
        printf("Arg[%d]? ", numargs);
        if ( fgets(argbuf, ARGLEN, stdin) && *argbuf != '\n' )
            arglist[numargs++] = makestring(argbuf);
        else
        {
            if ( numargs > 0 ){       /* any args? */
```

```
            arglist[numargs] = NULL;  /* close list */
            execute( arglist );       /* do it */
            numargs = 0;              /* and reset */
         }
      }
   }
   return 0;
}

execute( char *arglist[] )
/*
 * use fork and execvp and wait to do it
 */
{
   int pid,exitstatus;                /* of child */

   pid = fork();                      /* make new process */
   switch( pid ){
      case -1:
         perror("fork failed");
         exit(1);
      case 0:
         execvp(arglist[0], arglist);/* do it */
         perror("execvp failed");
         exit(1);
      default:
         while( wait(&exitstatus) != pid )
            ;
         printf ("child exited with status %d, %d\n",
                 exitstatus>>8, exitstatus&0377);
   }
}
char *makestring( char *buf )
/*
 * trim off newline and create storage for the string
 */
{
   char *cp, *malloc();

   buf[strlen(buf)-1] = '\0';         /* trim newline */
   cp = malloc( strlen(buf)+1 );      /* get memory */
   if ( cp == NULL ){                 /* or die */
      fprintf(stderr,"no memory\n");
      exit(1);
   }
```

```
    strcpy(cp, buf);                    /* copy chars */
    return cp;                          /* return ptr */
}
```

测试一下 psh2，看看它是否解决了只能运行一条命令这个问题：

```
$ ./psh2
Arg[0]? ls
Arg[1]? -l
Arg[2]? demodir
Arg[3]?
total 2
drwxr-x---    2 bruce    users     1024 Jul 14 21:02 a
drwxr-x---    3 bruce    users     1024 Jul 16 03:16 c
-rw-r--r--    1 bruce    users        0 Jul 14 21:03 y
child exited with status 0,0
Arg[0]? ps
Arg[1]?
  PID       TTY          TIME        CMD
11616       pts/4     00:00:00       bash
11648       pts/4     00:00:00       psh2
11664       pts/4     00:00:00       ps
child exited with status 0,0
Arg[0]? psh1   瞧！能运行 psh1 了
Arg[1]?
Arg[0]? ps     这是 psh1 的提示符！
Arg[1]?
PID      TTY        TIME      CMD
11616    pts/4      00:00:00  bash
11648    pts/4      00:00:00  psh2
11683    pts/4      00:00:00  ps
child exited with status 0,0
Arg[0]? grep
Arg[1]? fred
Arg[2]? /etc/passwd
Arg[3]?
child exited with status 1,0
Arg[0]? 按^D
Arg[0]? Arg[0]? Arg[0]? exit
Arg[1]?
execvp failed: No such file or directory
child exited with status 1,0
Arg[0]? 按^C
$
```

如何做到？

psh2.c 工作正常。新的 shell 接受程序名称、参数列表、运行程序、报告结果，然后再重新接受和运行其他程序。psh2.c 缺少常用的 shell 的一些修饰性功能，但可以作为一个坚实的基础开始了。

下一个版本中将做如下改进：

(1) 让用户可以通过按下 Ctrl－D 或者输入"exit"退出程序；

(2) 让用户能够在一行中输入所有参数。

在下一章实现的版本中加上这些功能。在那个版本中，将加上一些变量和控制流程使它更像一个编程语言。

这之前必须更正一个严重的错误。

信号和 psh2.c

从测试中可以看到，退出程序 psh2 的惟一方法是按 Ctrl－C 键。如果在 psh2 等待子进程结束时键入 Ctrl－C 键会如何呢？比如：

```
$ ./psh2
Arg[0]? tr
Arg[1]? [a - z]
Arg[2]? [A - Z]
Arg[3]?
hello
HELLO
now to press
NOW TO PRESS
Ctrl - C 按下^C
$
```

子进程结束，但是 shell 也结束了。按下 Ctrl－C 所生成的 SIGINT 信号不但杀死了运行 tr 的进程，而且杀死了运行 psh2 的进程。为什么呢？

键盘信号发给所有连接的进程

程序 psh2 和 tr 都连接到终端（如图 8.15 所示）。当按下中断键，ttr 驱动告诉内核向所

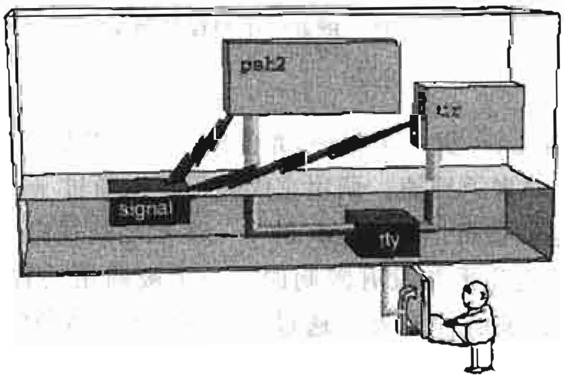


图 8.15　键盘信号发向所有连接的进程

有由这个终端控制的进程发送 SIGINT 信号。tr 死了，在我们的程序里，psh 也死掉了，即使它还在等待子进程的结束。

如何才能让 shell 不被用户按下的中断或退出键杀死？这个改动留作习题。

8.6 思考：用进程编程

为了理解 Unix 进程，运行了 ps 命令，还学习了 shell 如何使用 fork、exit 和 wait 来控制进程和运行程序。

在继续学习下一章有关在 shell 中增加变量和循环之前，考虑一下函数和进程之间的相似性。

1. execvp/exit 就像 call/return

(1) call/return

一个 C 程序由很多函数组成。一个函数可以调用另一个函数，同时传给它一些参数。被调用的函数执行一定的操作，然后返回一个值。每个函数都有它的局部变量，不同的函数通过 call/return 系统进行通信。

这种通过参数和返回值在拥有私有数据的函数间通信的模式是结构化程序设计的基础。Unix 鼓励将这种应用于程序之内的模式扩展到程序之间。这种模式可以用图 8.16 来表示。

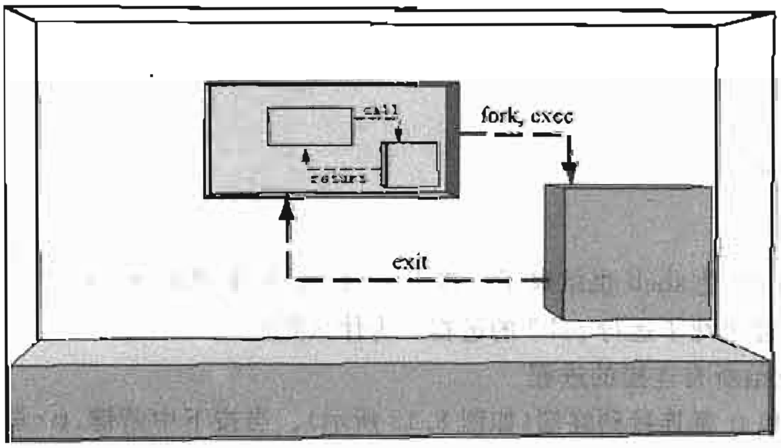


图 8.16　函数调用和程序调用

(2) exec/exit

一个 C 程序可以 fork/exec 另一个程序，并传给它一些参数。这个被调用的程序执行一定的操作，然后通过 exit(n)来返回值。调用它的进程可以通过 wait(&result)来获取 exit 的返回值。子程序的 exit 返回值可以在 result 的 8～15 位之间找到。

函数调用所用到的堆栈几乎是没有限制的。一个被调用的程序还可以调用其他程序，一个通过 fork/exec 调用起来的程序可以通过 fork/exec 调用别的程序。Unix 使创建一个新进程方便而且快捷。用 fork/exit 和 exit/wait 来调用程序和返回结果不仅适用于 shell，Unix 程序经常被设计成一组子程序，而不是一个带有很多函数的大程序。

由 exec 传递的参数必须是字符串。由于进程间通信的参数类型为字符串，这样就强迫了子程序的通信也必须使用文本作为参数类型。几乎是偶然的，这种基于文本的程序接口支持跨平台的交互，而这一点非常重要。

2. *全局变量和 fork/exec*

全局变量是有害的，它破坏了封装原则，导致出人意料的副作用[①]和难以维护的代码。但有时候去掉全局变量却更糟糕。怎么才能做到不将参数变得复杂的情况下管理一堆每个人都要用到的变量？尤其是必须将它们向下传几层的时候，情况会更加麻烦。

Unix 提供方法来建立全局变量。环境（environment）是一些传递给进程的字符串型变量集合。不会有副作用，它对 fork/exec 和 exit/wait 机制是一个有用的补充。下一章将看到它如何工作和如何使用它。

8.7　exit 和 exec 的其他细节

这一章主要学习进程、fork、execvp 和 wait，但是还需要再多了解一些关于 exit 和 exec 的细节。

8.7.1　进程死亡：exit 和_exit

exit 是 fork 的逆操作，进程通过调用 exit 来停止运行。fork 创建一个进程，exit 删除进程。基本上是这样。

exit 刷新所有的流，调用由 atexit 和 on_exit 注册的函数，执行当前系统定义的其他与 exit 相关的操作。然后调用_exit。系统函数_exit 是一个内核操作，这个操作处理所有分配给这个进程的内存，关闭所有这个进程打开的文件，释放所有内核用来管理和维护这个进程的数据结构。

子进程传给 exit 的参数被如何处理了？那个进程的弥留之言被存放在内核直到这个进程的父进程通过 wait 系统调用取回这个值。如果父进程没有在等这个值，那么它将被保存在内核直到父进程调用 wait，那时内核将通告这个父进程子进程的结束，并转达子进程的弥留之言。

那些已经死亡但是还没有给 exit 赋值的进程被称之为幽灵（zombie）进程。很多比较新的版本的 ps 列出这些进程并标记为 defunct。

_exit()小结如下。

系统调用_exit 终止当前进程并执行所有必须的清理工作。这些工作在各个不同版本的 Unix 中有些不同，但都包括以下一些操作：

(1) 关闭所有文件描述符和目录描述符。

(2) 将该进程的 PID 置为 init 进程的 PID。

(3) 如果父进程调用 wait 或 waitpid 来等待子进程结束，则通知父进程。

(4) 向父进程发送 SIGCHLD。

---

① 有些人认为全局变量会导致错误和混乱。

	_exit
目标	终止当前进程
头文件	#include <unistd.h> #include <stdlib.h>
函数原型	void _exit(int status)
参数	status 返回值
返回值	无
相关内容	atexit(3),exit(3),on_exit(3)

这样,如果父进程在子进程之前退出,那么子进程将能继续运行,而不会成为"孤儿",它们将是 init 进程的"子女",这有点像孤儿由国家监护。注意,就算父进程没有调用 wait,内核也会向它发送 SIGCHLD 消息。尽管对 SIGCHLD 消息的默认处理方法是忽略的。如果想响应这个消息,可以设置一个处理函数。

8.7.2 exec 家族

在已经实现的 shell 中和例程中用 execvp 来演示进程是如何运行一个程序的。execvp 不是一个系统调用,它是一个库函数,这个函数通过系统调用 execve 来调用内核服务。execve 中的 e 代表环境(environment),所以将推迟到下一章来讨论它。

还有一些调用 execve 的函数也是有用的。下面是这个家族中的一些成员:

```
execlp(file,argv0,argv1,...,NULL)
```

execlp 不像 execvp 那样用一个参数数组。相反,传给 main 的 argv[]中包括的参数被简单的放在 execlp 的参数中。例如:

```
execlp("ls","ls","-a","demodir",NULL);
```

以指定的参数运行程序 ls。当预先知道要运行的命令和它的参数时 execlp 是有用的。但是在 shell 中,这个函数没什么用,因为在用户输入命令之前不知道有多少参数。

```
execl(fullpath,argv0,argv1,...,NULL);
```

execlp 和 execvp 中的 p 代表路径(path)。这两个函数在环境变量 PATH 中列出的路径中查找由第一个参数指定的程序。如果准确知道这个文件的位置,那么就能够在 execl 中的第一参数中指定它的完整路径。例如:

```
execl("/bin/ls","ls","-a","demodir",NULL);
```

以指定的参数来运行程序/bin/ls。指定程序的准确位置比运行 execlp 更快。因为后者要在路径中查找指定程序。指定准确路径也比 execlp 安全。如果环境变量中有错误的路径列表,那么可能运行错误的程序。

```
execv(fullpath,arglist)
```

除了不在 PATH 中查找程序文件外，execv 和 execvp 非常相似。第一个参数必须是要执行程序的完整路径。使用 execv 或 execl 来执行明确指定的程序比依赖安全的路径列表 PATH 更安全。因为 PATH 很容易被恶意的用户篡改。

小　　结

1. 主要内容

- Unix 通过将可执行代码载入进程并执行它来运行一个程序。进程是运行一个程序所需的内存空间和其他资源的集合。
- 每个运行中的程序在自己的进程中运行。每个进程都有一个惟一的进程 ID、所有者、大小及其他属性。
- 系统调用 fork 通过复制进程来建立一个几乎和原来进程完全相同的副本进程。这个新建的进程被称为子进程。
- 一个程序通过调用 exec 函数族在当前进程中执行一个新的程序。
- 一个程序能通过调用 wait 来等待子进程结束。
- 调用程序能将一个字符串列表传给新程序的 main 函数。新的程序能通过调用 exit 来回传一个 8 位长的值。
- Unix shell 通过调用 fork、exec 和 wait 来运行程序。

2. 进一步的问题

shell 运行程序，同时 shell 也是一种编程语言。下面将学习 shell 的脚本语言。还要看看如何修改程序以支持脚本、控制逻辑和变量。

3. 习题

8.1　从 fork 返回的值能够区分父进程还是子进程？还有其他的办法做到这一点吗？

8.2　预测下面程序的输出：

```
main()
{
    int     n;
    for(n = 0; n<10 ; n++ )
    {
        printf("my pid = %d, n = %d\n", getpid() , n);
        sleep(1);
        if (fork() != 0)            /* what if these two */
            exit(0);                /* lines were removed */
    }
}
```

如果把有注解的两行删除，结果又会如何？

8.3　psh2.c 使用了定长的数组来存放参数列表。如何修改程序才能去掉用户输入命令参数个数的限制？这样的改动有必要吗？这就是说，Unix 是否限制了 exec 可

接受的参数长度或个数？

8.4　考虑以下代码：

```
main()
{   int fd;
    int pid;
    char msg1[] = "Test 1 2 3 ..\n";
    char msg2[] = "Hello, hello\n";

    if ( (fd = creat(tesefile, 0644)) == -1)
      return 0;
    if ( write(fd, msg1, strlen(msg1)) == -1)
      return 0;

    if ( (pid = fork()) == -1)
      return 0;
    if ( write(fd, msg2, strlen(msg2)) == -1)
      return 0;
    close(fd);
    return 1;
}
```

测试这个程序。调用 fork 之后两个进程都有一个指向同一个输出文件并且具有相同的当前位置的文件描述符。文件中会有几条记录？能否从记录的条数得知文件描述符和连接的文件？

8.5　考虑下面代码：

```
#include <stdio.h>
main()
{
    FILE *fp;
    int pid;
    char msg1[] = "Test 1 2 3 ..\n";
    char msg2[] = "Hello, hello\n";
    if ( (fp = fopen("testfile2", "w")) == NULL)
      return 0;
    fprintf(fp, "%s", msg1);
    if ( (pid = fork()) == -1)
      return 0;
    fprintf(fp,"%s",msgz);
    fclose(fp);
    return 1;
}
```

测试这个程序。文件中有多少条记录？解释一下结果。将这个程序的输出与课本中的 forkdemo1.c 的输出比较一下。

8.6　编译并运行以下程序：

```
main()
{
    int  i;
    if (fork() != 0)
      exit(0);
    for ( i = 1 ; i<= 10 ;i++ ){
       printf("still here..\n");
       sleep(i);
    }
    return 0;
}
```

解释程序做了些什么，怎么工作的，Unix shell 允许用户在后台运行程序。这个程序与后台进程有什么相似之处？

8.7　如果子进程运行失败，程序调用 exit。调用 exit 看起来很极端。为什么不仅仅从函数中返回出错代码？

4. 编程练习

8.8　扩展 waitdemo1.c 的功能，使之建立两个进程，并等待两个进程都结束。
进一步扩展你的程序使之能从命令行接受整数。然后程序创立该整数指定的进程数。分配每个进程一个随机的睡眠时间。最后，父进程报告每个子进程的退出。

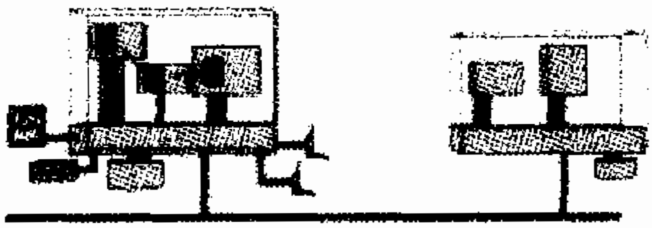
8.9　写个程序来帮助理解 SIGCHLD。修改 waitdemo2.c，设置 SIGCHLD 信号的处理函数。然后执行循环，每一秒打印一次"waiting"。当子进程退出，程序打印消息，报告退出原因然后退出。

8.10　写一个程序接受一个整数作为参数，然后创建参数指定个数的子进程。每个子进程睡眠 5 秒钟，然后退出。父进程设置 SIGCHLD 的信号处理函数。然后进入循环，每秒打印一次消息。信号处理函数调用 wait，然后打印子进程的 ID，最后将计数器增 1。当计数器达到建立的子进程数时，程序退出。用不同数目的子进程数来测试程序。当子进程数很大时程序可能会丢失一些子进程退出的消息。能解释为什么会有消息丢失吗？有没有办法解决？

8.11　修改 psh2.c 使之在用户输入"exit"或遇到文件结束时退出。

8.12　当有子进程在运行时，标准 Unix shell 在用户发送中断或退出信号时并不终止。接受命令行时，标准的 Unix shell 如何响应这些信号？修改 psh2.c，使之像一个标准的 shell 那样工作。

第 9 章　可编程的 shell、shell 变量和环境：编写自己的 shell



概念与技巧

- Unix shell 是一种编程语言
- 什么是 shell 脚本语言？shell 如何处理脚本语言？
- shell 如何处理结构化的工作？exit(0) = success
- 为什么需要 shell 变量以及如何使用 shell 变量
- 什么是环境？它是如何工作的？

相关的系统调用

- exit
- getenv

相关命令

- env

9.1　shell 编程

在 shell 中可以运行程序，而 shell 本身就是一种编程语言。shell 程序，一般称之为 shell 脚本，是 Unix 的重要部分，Unix 的引导程序和很多管理程序都使用 shell 脚本。本章中，首先学习 shell 的编程特征。然后在上一章编写的 shell 程序中增加一些特征，将 if..then 控制语句、局部变量和全局变量添加到要实现的 shell 程序中。

9.2　什么是以及为什么要使用 shell 脚本语言

shell 是一个编程语言解释器，这个解释器解释从键盘输入的命令，也解释存储在脚本中的命令序列。

shell 脚本包含一系列命令

shell 脚本是一个包含一系列命令的文件。运行一个脚本就是运行这个文件中的每个命

令。可以用一个 shell 脚本在一次请求中来执行多个命令。下面是一个例子：

```
#this is called script0
#it runs some commands
ls
echo the current date/time is
date
echo my name is
whoami
```

前两行是注释。shell 忽略以字符 # 开始的行,脚本的余下部分由命令组成,shell 逐条执行命令直到文件末尾或者 shell 执行到 exit 命令。

可以把脚本文件名作为参数传给 shell 来执行脚本：

```
$ sh script0
script0 script1 script2 script3
the current date/time is
Sun Jul 29 23:29:49 EDT 2001
my name is
bruce
$
```

还可以通过设置文件的执行权限,然后输入文件名来执行脚本：

```
$ chmod +x script0
$ script0
script0 script1 script2 script3
the current date/time is
Sun Jul 29 23:31:23 EDT 2001
my name is
bruce
$
```

对于一个脚本只需要执行一次 chmod,可执行位将保持不变直到下一次再改变它。用第二种方法,也就是用改变文件可执行属性的方法来启动脚本会更加方便。将脚本设置成可执行的,然后像运行系统命令或自己编写的程序一样来执行脚本。

将使用哪个 shell? 我们将学习和编写脚本的 shell 是一个早期版本的 Unix shell:sh 的语法,它被称为 B shell(Bourne Shell),这是根据编写这个程序的人的名字来命名的。过去的几年中有很多不同的 shell 被实现,它们各有各的特点或语法。这里将学习的语法在大多数 shell 中是相同的,包括 sh、bash 和 ksh。

1. sh 的编程特征:变量、I/O 和 if..then

shell 脚本是真正的程序。注意在 script2 中体现的特点：

```
#! /bin/sh
# script2 : a real program with variables, input,
#           and control flow

BOOK = $HOME/phonebook.data
echo find what name in phonebook
read NAME
if grep $ NAME $ BOOK > /tmp/pb.tmp
then
   echo Entries for $ NAME
   cat /tmp/pb.tmp
else
   echo No entries for $ NAME
fi
rm /tmp/pb.tmp
```

下面是 script2 的输出：

```
$ ./script2
find what name in phonebook
dave
Entries for dave
dave     432 - 6546
$ ./script2
find what name in phonebook
fran
No entries for fran
$ cat $HOME/phonebook.data
ann      222 - 3456
bob      323 - 2222
carla    123 - 4567
dave     432 - 6546
eloise   567 - 9876
$
```

脚本中除了命令之外还包括以下元素。

(1) 变量

脚本中可以定义变量。在 script2 中，定义了名为 BOOK 和 NAME 两个变量，并在定义之后使用了它们，用前缀$ 来取得变量的值。变量名不一定要大写，只是习惯上将其大写。

(2) 用户输入

read 命令告诉 shell 要从标准输入中读入一个字符串。可以使用 read 来创建交互的脚本，也可以从文件或管道中读入数据。

(3) 控制

这个脚本包括了 if..then..else..fi 控制语句。其他的脚本控制语句还有 while、case

和 for。

(4) 环境

脚本使用一个名为 HOME 的变量。HOME 的值是你的主目录的路径。HOME 变量是由 login 程序设置的，可以被 login 进程的所有子进程使用。HOME 变量是多个环境变量(environment variables)中的一个。这些环境变量记录了个性化设置。而这些设置能影响很多程序的行为。比如，TZ 变量记录了当前的时区。将 TZ 设置为"EST5EDT"是告诉那些使用 ctime 的程序，比如 date 或 ls -l，应该显示美国东部时间。本章后面会学习环境变量的作用和结构。

2. 自编 shell 的改进

在上一章用 fork、execvp 和 wait 实现了一个能够创建进程和运行程序的 shell。本章中，将对这个 shell 做一些改进。首先，将加入命令行解析。这样用户就能够在一行中输入命令和所有参数了。然后，将控制语句 if..then 加入到这个 shell 中。最后将加入局部变量和环境变量。

9.3 smsh1——命令行解析

对自编 shell 的第一个改进是添加命令行解析的功能。这个版本命名为 smsh1.c。用户现在可以在一行中输入，比如：

```
find /home -name core -mtime +3 -print
```

然后由解析器将命令行拆成字符串数组，以便传给 execvp。程序主要流程如图 9.1 所示。对 psh2.c 的改进包括将命令行分解成参数数组，在 shell 中忽略信号 SIGINT 和 SIGQUIT，但是在子进程中恢复对信号 SIGINT 和 SIGQUIT 的默认操作，允许用户通过按表示结束文件的 Ctrl-D 键来退出。

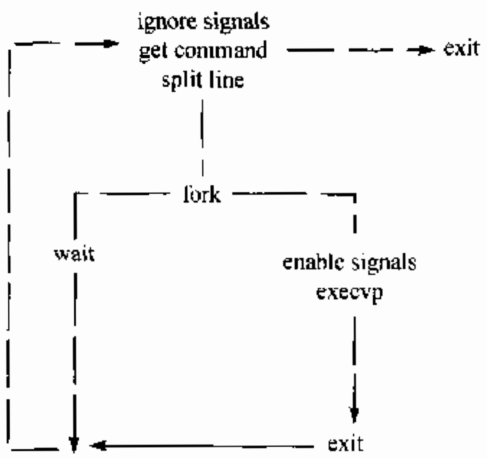


图 9.1　一个有信号、退出和解析的 shell

shell 的主函数如下：

```
int main()
{
   char     * cmdline, * prompt, * * arglist;
   int      result;
   void     setup();

   prompt = DFL_PROMPT ;
   setup();

   while ( (cmdline = next_cmd(prompt, stdin)) != NULL ){
      if ( (arglist = splitline(cmdline)) != NULL ){
         result = execute(arglist);
         freelist(arglist);
      }
      free(cmdline);
   }
   return 0;
}
```

3 个函数的解释如下。

(1) next_cmd

next_cmd 从输入流中读入下一个命令。它调用 malloc 来分配内存以接受任意长度的命令行。碰到文件结束符，它返回 NULL。

(2) splitline

splitline 将一个字符串分解为字符串数组，并返回这个数组。它调用 malloc 来分配内存以接受任意参数个数的命令行。这个数组由 NULL 标记结束。

(3) execute

execute 使用 fork、execvp 和 wait 来运行一个命令。execute 返回命令的结束状态。

smsh1 由 3 个文件组成：smsh1.c、splitline.c 和 execute.c。用以下命令来编译和运行这个程序：

```
$ cc smsh1.c splitline.c execute.c -o smsh1
$ ./smsh1
>ps -f
UID     PID     PPID    C    STIME   TTY     TIME        CMD
bruce   23203   23199   0    Jul29   pts/4   00:00:00    bash
bruce   25383   23203   0    08:23   pts/4   00:00:00    ./smsh1
bruce   25385   25383   0    08:23   pts/4   00:00:00    ps -f
>在这里按 Ctrl-D 键
$
```

注意 ps -f 是./smsh1 的子进程。./smsh1 是 bash 的子进程。下面是 smsh1.c 的代码：

```
/**     smsh1.c small - shell version 1
 **     first really useful version after prompting shell
 **     this one parses the command line into strings
 **     uses fork, exec, wait, and ignores signals
 **/

#include    <stdio.h>
#include    <stdlib.h>
#include    <unistd.h>
#include    <signal.h>
#include    "smsh.h"

#define  DFL_PROMPT   "> "

int main()
{
   char     * cmdline, * prompt, * * arglist;
   int      result;
   void     setup();

   prompt = DFL_PROMPT ;
   setup();

   while ( (cmdline = next_cmd(prompt, stdin)) != NULL ){
      if ( (arglist = splitline(cmdline)) != NULL ){
         result = execute(arglist);
         freelist(arglist);
      }
      free(cmdline);
   }
   return 0;
}

void setup()
/*
 * purpose: initialize shell
 * returns: nothing. calls fatal() if trouble
 */
{
   signal(SIGINT, SIG_IGN);
   signal(SIGQUIT, SIG_IGN);
}

void fatal(char * s1, char * s2, int n)
{
   fprintf(stderr,"Error: %s, %s\n", s1, s2);
   exit(n);
```

```
}
```

下面是 execute.c 的代码：

```
/* execute.c - code used by small shell to execute commands */

#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <signal.h>
#include  <sys/wait.h>

int execute(char *argv[])
/*
 * purpose: run a program passing it arguments
 * returns: status returned via wait, or -1 on error
 * errors: -1 on fork() or wait() errors
 */
{
    int    pid ;
    int    child info = -1;

    if ( argv[0] == NULL )     /* nothing succeeds*/
        return 0;

    if ( (pid = fork()) == -1 )
        perror("fork");
    else if ( pid == 0 ){
        signal(SIGINT, SIG_DFL);
        signal(SIGQUIT, SIG_DFL);
        execvp(argv[0], argv);
        perror("cannot execute command");
        exit(1);
}
else {
    if ( wait(&child_info) == -1 )
        perror("wait");
    }
    return child_info;
}
```

下面是 splitline.c 的代码：

```
/* splitline.c - commmand reading and parsing functions for smsh
 *
 *    char *next_cmd(char *prompt, FILE *fp) - get next command
 *    char **splitline(char *str); - parse a string
```

```
 */

#include   <stdio.h>
#include   <stdlib.h>
#include   <string.h>
#include   "smsh.h"

char * next_cmd(char * prompt, FILE * fp)
/*
 * purpose: read next command line from fp
 * returns: dynamically allocated string holding command line
 * errors: NULL at EOF (not really an error)
 *         calls fatal from emalloc()
 * notes: allocates space in BUFSIZ chunks.
 */
{
    char    * buf ;                     /* the buffer */
    int     bufspace = 0;               /* total size */
    int     pos = 0;                    /* current position */
    int     c;                          /* input char */

    printf("%s", prompt);               /* prompt user */
    while( ( c = getc(fp)) != EOF ) {

        /* need space? */
        if ( pos + 1 >= bufspace ){     /* 1 for \0 */
            if ( bufspace == 0 )        /* y: 1st time */
                buf = emalloc(BUFSIZ);
            else/* or expand */
                buf = erealloc(buf,bufspace + BUFSIZ);
            bufspace += BUFSIZ;         /* update size */
        }

        /* end of command? */
        if ( c == '\n' )
            break;

        /* no, add to buffer */
        buf[pos++] = c;
    }
    if ( c == EOF && pos == 0 )         /* EOF and no input */
        return NULL;                    /* say so */
    buf[pos] = '\0';
    return buf;
}

/**
```

```
 ** splitline ( parse a line into an array of strings )
 ** /
#define  is_delim(x) ((x) == ' '||(x) == '\t')

char ** splitline(char * line)
/*
 * purpose: split a line into array of white - space separated tokens
 * returns:a NULL - terminated array of pointers to copies of the
 *           tokens or NULL if line if no tokens on the line
 * action: traverse the array, locate strings, make copies
 * note: strtok() could work, but we may want to add quotes later
 */
{
    char   * newstr();
    char   ** args ;
    int     spots = 0;                  /* spots in table   */
    int     bufspace = 0;               /* bytes in table   */
    int     argnum = 0;                 /* slots used       */
    char    * cp = line;                /* pos in string      */
    char    * start;
    int     len;

    if ( line == NULL )                 /* handle special case*/
        return NULL;

    args      = emalloc(BUFSIZ);        /* initialize array*/
    bufspace = BUFSIZ;
    spots     = BUFSIZ/sizeof(char * );

    while( * cp != '\0' )
    {
        while ( is_delim( * cp) )       /* skip leading spaces   */
            cp ++ ;
        if ( * cp == "\0" )             /* quit at end - o - string   */
            break;

        /* make sure the array has room ( + 1 for NULL) */
        if ( argnum + 1 >= spots ){
            args = erealloc(args,bufspace + BUFSIZ);
            bufspace += BUFSIZ;
            spots += (BUFSIZ/sizeof(char * ));
        }

        /* mark start, then find end of word */
        start = cp;
        len = 1;
        while ( * ++ cp != '\0' && ! (is_delim( * cp)) )
```

```
            len++;
        args[argnum++] = newstr(start, len);
    }
    args[argnum] = NULL;
    return args;
}

/*
 * purpose: constructor for strings
 * returns: a string, never NULL
 */
char *newstr(char *s, int l)
{
    char *rv = emalloc(l+1);

    rv[l] = '\0';
    strncpy(rv, s, l);
    return rv;
}

void freelist(char **list)
/*
 * purpose: free the list returned by splitline
 * returns: nothing
 * action: free all strings in list and then free the list
 */
{
    char    **cp = list;
    while( *cp )
        free( *cp++ );
    free(list);
}

void * emalloc(size_t n)
{
    void *rv ;
    if ( (rv = malloc(n)) == NULL )
        fatal("out of memory","",1);
    return rv;
}
void * erealloc(void *p, size_t n)
{
    void *rv;
    if ( (rv = realloc(p,n)) == NULL )
        fatal("realloc() failed","",1);
    return rv;
```

```
}
```

下面是 smsh.h 的代码：

```
#define    YES 1
#define    NO  0

char   *next_cmd();
char   **splitline(char *);
void   freelist(char **);
void   *emalloc(size_t);
void   *erealloc(void *, size_t);
int    execute(char **);
void   fatal(char *, char *, int);
```

关于 smsh1 的解释

smsh1 比 psh2 要好用很多，改进的方面主要包括以下一些：

(1) 一行多个命令

通常的 shell 允许用分号分隔命令，这样用户就可以在一行中输入多个命令了：

```
ls demodir; ps -f ; date
```

(2) 后台进程

通常的 shell 允许用户通过在命令的结尾加上与符号(&)来使其在后台运行，如：

```
find /home -name core  print &
```

在后台运行一个程序意味着一旦启动它，系统将立即返回提示符，这个进程可能还没有结束，但已经可以在提示符后输入命令并运行其他进程了。这听起来有点复杂，实际上实现是非常简单。看看流程图，想想是如何不等待命令结束而返回提示符的。想法很简单而且漂亮，但是要处理信号和防止僵尸(Zombies)进程，这有点像惊险片。

(3) 退出命令

通常的 shell 允许用户通过输入 exit 来退出 shell。exit 命令接受一个整数参数，比如 exit 3，这种情况下，这个数字被作为参数传给 exit 函数。

9.4　shell 中的流程控制

对原有的 shell 的第 2 个改进是增加 if..then 控制语句。

9.4.1　if 语句做些什么

shell 提供 if 控制语句。假设你计划每周五做磁盘备份。考虑一下以下例子：

```
if date|grep Fri
then
```

```
    echo time for backup. Insert tape and press enter
    read x
    tar cvf /dev/tape /home
fi
```

shell 中的 if 语句的作用与其他语言的 if 语句相同：条件检测。如果条件的值为正，则有一部分代码被执行。在 shell 中，条件是一个命令，返回正值意味着命令运行成功。

在例子中，命令是 date|grep Fri，这个命令在 date 的输出字符串中查找"Fri"字符串，如果找到则命令成功，否则命令失败。程序是如何表示成功的呢？

(1) exit(0)代表成功

grep 程序调用函数 exit(0)来表明成功。所有的 Unix 程序都遵从以 0 退出表明成功这一惯例。比如，diff 命令用来比较两个文本文件。如果两个文件相同，diff 返回 0 以表明成功。类似地，mv、cp 和 rm 都以相同的方式表明成功。脚本中的 if..then 语句基于以 0 退出表示成功这个假设。

(2) 带有 else 的 if 语句

一个 if 语句可以有 else 部分，比如：

```
ls
who
if diff file1 file1.bak
then
    echo no differences found, removing backup
    rm file1.bak
else
    echo backup differs, making it read-only
    chmod -w file1.bak
fi
date
```

else 部分就像 then 部分一样，可以包含任意数量的命令，包括其他的 if..then 语句。

if 语句还有另一个特征。如果 if 后的条件是一系列的命令，那么最后一个命令的 exit 值被用作这个语句块的条件值，并由此来决定条件是否成立。

9.4.2 if 是如何工作的

if 语句的工作流程主要如下。

(1) shell 运行 if 之后的命令。

(2) shell 检查命令的 exit 状态。

(3) exit 的状态为 0 意味着成功，非 0 意味着失败。

(4) 如果成功，shell 执行 then 部分的代码。

(5) 如果失败，shell 执行 else 部分的代码。

(6) 关键字 fi 标识 if 块的结束。

9.4.3 在 smsh 中增加 if

现在已经知道 if 控制语句做什么，也知道它是如何工作的。那么如何在 shell 中增加 if 语句呢？

这里已经知道如何运行一个命令——调用 execute。也知道如何检查一个程序的退出状态——从 wait 函数中得到。需要将 if 之后命令的结果存放在一些变量中，然后要知道后面读入的命令是在 then 块中，还是 else 块中。最后还得确保在 if 之后读入 then。

(1) 增加一层：process

要实现这些功能，原来的模型就有些太简单。smsh1 的控制流从 splitline 直接到 fork。每个命令都被直接传给 exec。新的版本中，以 if、then 或者 fi 开始的行和条件失败时 then 语句块中的命令行不传给 exec。添加 if 语句后使命令处理变得复杂，所以要写一个名为 process 的函数来包含这些复杂的代码。修改后的流程图如图 9.2 所示。

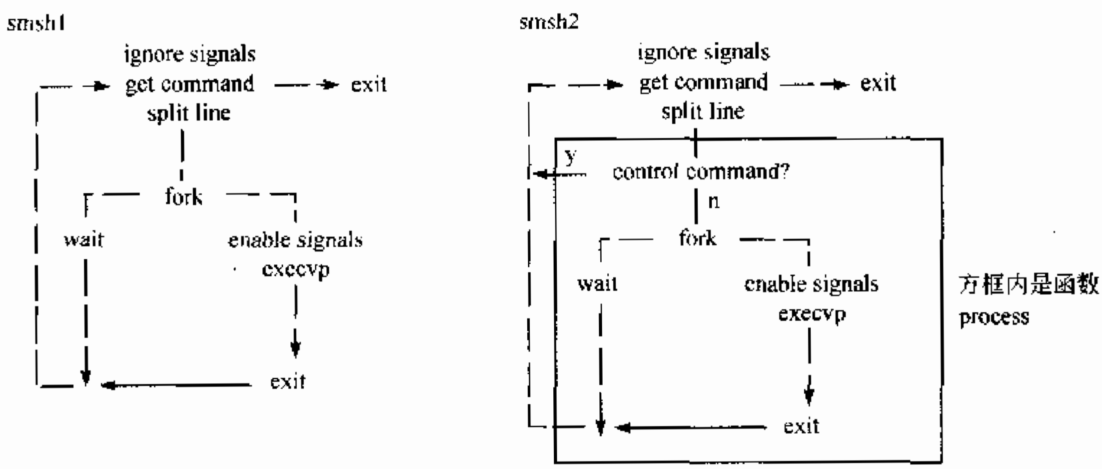


图 9.2 在 smsh 中增加流程控制

(2) process 做些什么

process 通过寻找关键字，比如 if、then 和 fi，来管理脚本流程，在适当的时候调用 fork 和 exec。process 必须记录条件命令的结果以便能够处理 then 和 else 块。

(3) procss 是如何工作的？代码区域、运行状态

process 将脚本看作一个接一个的代码区域。第 1 个区域是 then 代码块，第 2 个是 else 代码块，第 3 个是在 if 语句之外的代码块。就像图 9.3 所示，对于不同的区域，shell 的处理方法也是不同的。

考虑 if 语句之外的区域，这里称之为中立区(neutral)。对于这类区域的代码，简单地读一条，分析一条，执行一条。

接下来是在 if 和 then 之间的区域。这个区域中，shell 每执行一条命令就记录下它的退出状态，另一个区域是从 then 到 fi 或 else 之间，最后一个区域是从 else 到 fi，在 fi 之后又回到中立区了。

shell 记录当前区域类型，还必须记录在 WANT_THEN 区域中所执行命令的结果。

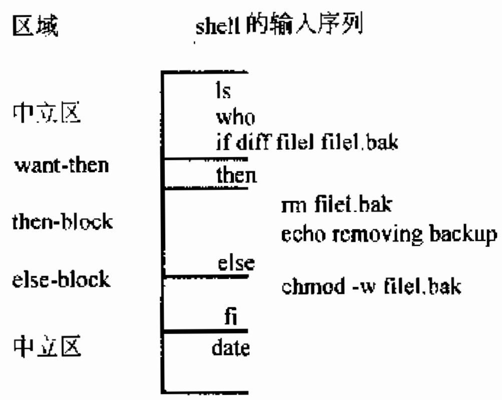


图 9.3　由不同区域组成的脚本

不同区域的处理方法是不同的。特定的区域与程序的特定状态联系在一起。process 通过 3 个函数来处理区域问题。

(1) is_control_command

is_control_command 返回一个 boolean 变量告诉 process 这条命令是脚本语言的一部分还是一条可执行的命令。

(2) do_control_command

do_control_command 处理关键字 if、then 和 fi。每个关键字都是区域的界标。这个函数更新状态变量并执行必要的操作。

(3) ok_to_execute

ok_to_execute 根据当前的状态和条件命令的结果返回一个 boolean 值，说明能否执行当前命令。

9.4.4　smsh2.c:修改后的代码

smsh2.c 是基于 smsh1.c 的。main 函数只有一处需要改动——调用 execute 的地方调用 process 了：

```
/** smsh2.c - small - shell version 2
 **     small shell that supports command line parsing
 **     and if..then..else.fi logic (by calling process())
 **/
#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <signal.h>
#include  <sys/wait.h>
#include  "smsh.h"

#define  DFL_PROMPT  "> "

int main()
```

```
{
    char  * cmdline, * prompt, ** arglist;
    int   result, process(char ** );
    void  setup();

    prompt = DFL_PROMPT ;
    setup();

    while ( (cmdline = next_cmd(prompt, stdin)) != NULL ){
        if ( (arglist = splitline(cmdline)) != NULL ){
            result = process(arglist);
            freelist(arglist);
        }
        free(cmdline);
    }
    return 0;
}

void setup()
/*
 * purpose: initialize shell
 * returns: nothing. calls fatal() if trouble
 */
{
    signal(SIGINT, SIG_IGN);
    signal(SIGQUIT, SIG_IGN);
}

void fatal(char * s1, char * s2, int n)
{
    fprintf(stderr,"Error: %s, %s\n", s1, s2);
    exit(n);
}
```

还添加了两个新文件，process.c 和 controlflow.c：

```
/* process.c
 * command processing layer
 *
 * The process(char ** arglist) function is called by the main loop
 * It sits in front of the execute() function. This layer handles
 * two main classes of processing:
 * a) built - in functions (e.g. exit(), set, = , read, .. )
 * b) control structures (e.g. if, while, for)
 */

#include  <stdio.h>
```

```
#include   "smsh.h"

int is_control_command(char * );
int do_control_command(char ** );
int ok_to_execute();

int process(char ** args)
/*
 * purpose: process user command
 * returns: result of processing command
 * details: if a built - in then call appropriate function, if not
            execute()
 * errors: arise from subroutines, handled there
 */
{
    int     rv = 0;

    if ( args[0] == NULL )
        rv = 0;
    else if ( is_control_command(args[0]) )
        rv = do_control_command(args);
    else if ( ok_to_execute() )
        rv - execute(args);
    return rv;
}
/* controlflow.c
 *
 * "if" processing is done with two state variables
 * if_state and if_result
 */
#include   <stdio.h>
#include   "smsh.h"

enum states   { NEUTRAL, WANT_THEN, THEN_BLOCK };
enum results  { SUCCESS, FAIL };

static int if_state = NEUTRAL;
static int if_result = SUCCESS;
static int last_stat = 0;

int  syn_err(char * );

int ok_to_execute()
/*
 * purpose: determine the shell should execute a command
 * returns: 1 for yes, 0 for no
 * details : if in THEN_BLOCK and if_result was SUCCESS then yes
```

```
 *        if in THEN_BLOCK and if_result was FAIL then no
 *        if in WANT_THEN then syntax error (sh is different)
 */
{
    int  rv = 1;     /* default is positive */

    if ( if_state == WANT_THEN ){
        syn_err("then expected");
        rv = 0;
    }
    else if ( if_state == THEN_BLOCK && if_result == SUCCESS )
        rv = 1;
    else if ( if_state == THEN_BLOCK && if_result == FAIL )
        rv = 0;
    return rv;
}

int is_control_command(char * s)
/*
 * purpose: boolean to report if the command is a shell control command
 * returns: 0 or 1
 */
{
    return (strcmp(s,"if") == 0 || strcmp(s,"then") == 0 ||
            strcmp(s,"fi") == 0);
}

int do_control_command(char ** args)
/*
 * purpose: Process "if", "then", "fi" - change state or detect error
 * returns: 0 if ok, -1 for syntax error
 */
{
    char  *cmd = args[0];
    int rv = -1;

    if ( strcmp(cmd,"if") == 0 ){
        if ( if_state != NEUTRAL )
            rv = syn_err("if unexpected");
        else {
            last_stat = process(args + 1);
            if_result = (last_stat == 0 ? SUCCESS : FAIL );
            if_state = WANT_THEN;
            rv = 0;
        }
    }
```

```
    else if ( strcmp(cmd,"then") == 0 ){
        if ( if_state != WANT_THEN )
            rv = syn_err("then unexpected");
        else {
            if_state = THEN_BLOCK;
            rv = 0;
        }
    }
    else if ( strcmp(cmd,"fi") == 0 ){
        if ( if_state != THEN_BLOCK )
            rv = syn_err("fi unexpected");
        else {
            if_state = NEUTRAL;
            rv = 0;
        }
    }
    else
        fatal("internal error processing:", cmd, 2);
    return rv;
}

int syn_err(char *msg)
/* purpose: handles syntax errors in control structures
 * details: resets state to NEUTRAL
 * returns: -1 in interactive mode. Should call fatal in scripts
 */
{
    if_state = NEUTRAL;
    fprintf(stderr,"syntax error: %s\n", msg);
    return -1;
}
```

在 controlflow.c 中，if 语句的 else 没有被处理，这一部分留作读者的习题。

编译并执行这个版本：

```
$ cc -o smsh2 smsh2.c splitline.c execute.c process.c controlflow.c
$ ./smsh2
> grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> if grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> then
>  echo ok
ok
> fi
```

```
> if grep pati /etc/passwd
> then
>   echo ok
> fi
> echo ok
ok
> then
syntax error: then unexpected
```

做得如何？

看起来做得不错。那么和常用的 shell 比起来又如何呢？

```
$ if grep lp /etc/passwd
$ then
$   echo ok
$ fi
lp:x:4:7:lp:/var/spool/lpd:
ok
$
```

这个 shell 处理 if 语句的方法和刚才的实现有些不同。标准的 shell 在读到 fi 后，整个地执行 if 语句块。这是怎么做到的呢？为什么要这么做呢？常用的 shell 还允许嵌套的 if 语句，这里的程序能做修改以支持嵌套的 if 语句吗？

9.5 shell 变量：局部和全局

像其它的程序语言一样，Unix shell 也有变量。能对这些变量赋值，也可从这些变量取值，列出所有变量，例如以下的代码：

```
$ age = 7                            # assigning a value
$ echo $ age                         # retrieving a value
7
$ echo age                           # the $  is required
age
$ echo $ age + $ age                 # purely string operations
7 + 7
$ read name                          # input from stdin
fido
$ echo hello, $ name, how are you    # can be interpolated
hello, fido, how are you
$ ls > $ name.$ age                  # used as a part of a command
$ food = muffins                     # no spaces in assignment
food:not found
$
```

shell 包括两类变量：局部变量和环境变量。在前面提到过了，像 HOME 和 TZ 这样的变量可以让用户把个性化设置传递给程序，这些变量的作用有点象全局变量，它们可以被所有 shell 的子进程存取。本章的后面将深入了解环境，现在，只要记住有两类变量就可以了。

9.5.1 使用 shell 变量

前面的例子演示了对变量的大部分操作。对变量的操作如下。

操作类型	语 法	注 释
赋值	var=value	不能有空格
引用	$var	
删除	unset var	
输入	read var	也可以 read var1 var2...
列出变量	set	
全局化	export var	

变量名是字符 A～Z、a～z、0～9 和_的组合。第一个字母不能是数字。变量名是大小写敏感的。

变量的值是字符串。变量都是字符串类型的，没有数值类型的变量。所有的操作都是字符串操作。

列出所有变量使用 set 命令。set 命令列出当前 shell 定义的所有变量，例如：

```
$ set
BASH=/bin/bash
BASH_VERSION=1.14.7(1)
DISPLAY=:0.0
EUID=500
HOME=/home2/bruce
HOSTTYPE=i386
IFS=

LANG=en
LANGUAGE=en
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib
LOGNAME=bruce
OPTERR=1
OPTIND=1
OSTYPE=Linux
PATH=/bin:/usr/bin:/usr/X11R6/bin::/usr/local/bin:/home2/bruce/bin
PPID=30928
PS4=+
PWD=/home2/bruce/projs/ubook/src/ch09
SHELL=/bin/bash
```

```
SHLVL = 2
TERM = xterm - color
UID = 500
USER = bruce
_ = /bin/vi
age = 7
name = fido
```

这个列表中包括很多在登录时设置的变量，加上两个后面新增的局部变量。

9.5.2　变量的存储

要在 shell 里增加变量，必须有个地方能存放这些变量的名称和值，而且这个变量存储系统必须能够分辨局部和全局变量。下面是这个存储系统的抽象模型：

(1) 模型

变量	值	是否为全局变量?
data	"phonebook.dat"	n
HOME	"/home2/fido"	y
TERM	"t1061"	y

(2) 接口(部分)

VLstore(char * var,char * val) 增加/更新 var=val

VLookup(char * var) 取得 var 的值

VList 输出列表到 stdout

(3) 实现

可以用链表、hash 表、树或者是几乎任何数据结构来实现它。作为第一个版本，用一个结构数组，其中的每个变量是这样的结构：

```
struct var{
   char * str;          /*name = val string*/
   int global;          /*a boolean*/
};
static struct var tab[MAXVARS];
```

如图 9.4 所示。

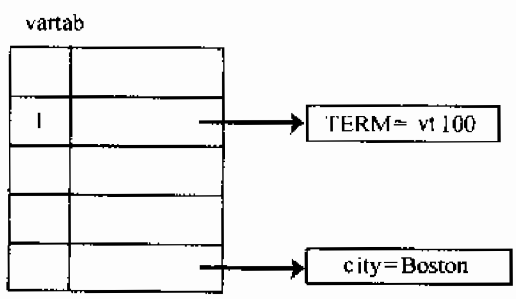


图 9.4　shell 变量的存储方式

9.5.3　增加变量命令：Built-ins

已经有地方存放变量了，但还要增加给变量赋值、列出所有变量和获取变量值的命令。这就是说，在这个 shell 中，用户应该能够输入：

```
>TERM = xterm
```

```
>set
>echo $ TERM
```

set 是 shell 的一个命令，而不是一个由 shell 运行的程序，这就像 if 和 then 这些关键字由 shell 自己处理一样。为了将 set 与要被执行的程序区分开，将 set 设置为内置(built－in)的命令。

命令 varname＝value 告诉 shell 在变量表里添加一项，赋值语句也是内置的命令。

为了增加内置的命令到这个 shell 中，需要对流程图做另外的一些修改。在调用 fork 和 exec 之前必须先看看命令是否是 shell 内置的命令，如图 9.5 所示。

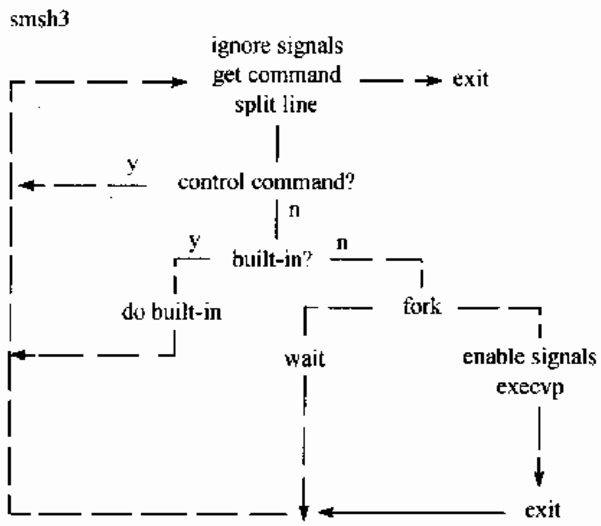


图 9.5　向 smsh 中添加内置命令

修改 process 函数，使之在调用 fork/exec 之前检查是否为内置的命令：

```
if ( args[0] == NULL )
    rv = 0;
else if ( is_control_command(args[0]) )
    rv = do_control_command(args);
else if ( ok_to_execute() ){
    if ( ! builtin_command(args,&rv) )
        rv = execute(args);
}
```

新的函数 builtin_command 将检查和执行内置命令合并在一起。变量 rv 用来标识状态，builtin_command 返回一个布尔值，并修改状态变量 rv。

builtin.c 的代码如下：

```
/* builtin.c
 * contains the switch and the functions for builtin commands
 */
```

```
#include  <stdio.h>
#include  <string.h>
#include  <ctype.h>
#include  "smsh.h"
#include  "varlib.h"

int assign(char *);
int okname(char *);

int builtin_command(char **args, int *resultp)
/*
 * purpose: run a builtin command
 * returns: 1 if args[0] is builtin, 0 if not
 * details: test args[0] against all known built-ins. Call functions
 */
{
    int rv = 0;

    if ( strcmp(args[0],"set") == 0 ){              /* 'set' command? */
        VLlist();
        *resultp = 0;
        rv = 1;
    }
    else if ( strchr(args[0], '=') != NULL ){       /* assignment cmd */
        *resultp = assign(args[0]);
        if ( *resultp != -1 )                       /* x-y=123 not ok */
            rv = 1;
    }
    else if ( strcmp(args[0], "export") == 0 ){
        if ( args[1] != NULL && okname(args[1]) )
            *resultp = VLexport(args[1]);
        else
            *resultp = 1;
        rv = 1;
    }
    return rv;
}

int assign(char *str)
/*
 * purpose: execute name=val AND ensure that name is legal
 * returns: -1 for illegal lval, or result of VLstore
 * warning: modifies the string, but retores it to normal
 */
{
    char    *cp;
```

```
    int   rv ;

    cp = strchr(str,'=');
    *cp = '\0';
    rv = ( okname(str) ? VLstore(str,cp+1) : -1 );
    *cp = '=';
    return rv;
}
int okname(char *str)
/*
 * purpose: determines if a string is a legal variable name
 * returns: 0 for no, 1 for yes
 */
{
    char   *cp;

    for(cp = str; *cp; cp++ ){
    if ( (isdigit(*cp) && cp == str) || !(isalnum(*cp) || *cp == '_' ))
            return 0;
    }
    return ( cp != str );/* no empty strings, either */
}
```

9.5.4 效果如何

编译并运行改进后的程序:

```
$ cc -o smsh3.c smsh2.c splitline.c execute.c process2.c \
controlflow.c builtin.c varlib.c
$ ./smsh3
>set
>day = monday
>temp = 75
>TZ = CST6CDT
>x.y = z
cannot execute command:No such file or directory
> set
    day = monday
    temp = 75
    TZ = CST6CDT
> date
Tue Jul 31 11:56:59 EDT 2001
> echo $ temp, $ day
$ temp, $ day
```

(1) 已经可以正常工作了

这里的 shell 现在支持变量了，能够给变量赋值，也可以列出当前的变量，程序甚至会检查不合法的变量名，不会将这些名字作为程序名来处理。

(2) TZ 没有传给 Date

从这里的例子看出还有两件事要做。先将变量 TZ 的值设为美国中部时间(U. S. central time)，但是 date 命令报告的还是美国东部时间。前面说过，变量 TZ 是程序运行环境的一部分，它的值应该从父进程传给子进程，这如何才能实现呢？这里的 shell 如何才能把变量放到环境中使它的子进程能够访问得到？所以，接下来将讨论环境。

(3) 变量$ temp 和$ day 的值没有被正确显示

刚才的测试表明这两个变量的值没有被 shell 正确显示，也就是说，当 shell 在处理 echo $ temp，$ day 时没有用变量的值替换变量名。这些变量是 shell 的局部变量，echo 命令不知道这些变量的值，所以 shell 在执行外部程序之前必须进行变量替换。本章的末尾将会再探究这个问题。

9.6 环境:个性化设置

人们喜欢按照自己的喜好设置自己的电脑，有些人喜欢用风景画作桌面，而其他一些人可能更喜欢纯色的桌面，有些人喜欢用 emacs 来编辑文本，而有些人喜欢 vi。Unix 允许用户在称之为环境(environment)的地方以变量的形式存放这些设置。每个用户有一个惟一的主目录、用户名、邮件文件、终端类型和喜欢用的编辑器，很多个性化的设置由环境中的变量记录。

很多程序的行为基于这些设置，比如，运行 script3，可以看到 date 根据 TZ 值的不同显示不同的格式：

```
#! /bin/sh
# script3 - shows how an environment variable is passed to commands
# TZ is time zone, affect things like date, and ls - l
#
echo "The time in Boston is"
   TZ = EST5EDT
   export TZ                 # add TZ to the environment
   date                      # date uses the value in TZ
echo "The time in Chicago is"
   TZ = CST6CDT
   date
echo "The time in LA is"
   TZ = PST8PDT
   date
```

环境不是 shell 的一部分。但是 shell 包括一些可以让用户读取和修改环境的命令。一如既往，先瞧瞧环境做些什么，然后学习它是如何工作的，最后把它加到实现的代码中。

9.6.1 使用环境

1. 列出环境

env 命令列出当前所有环境设置：

```
$ env
LOGNAME = bruce
LD_LIBRARY_PATH = /usr/lib:/usr/local/lib
TERM = xterm - color
HOSTTYPE = i386
PATH = /bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:/home2/bruce/bin
HOME = /home2/bruce
SHELL = /bin/bash
USER = bruce
LANGUAGE = en
DISPLAY = :0.0
LANG = en
_ = /usr/bin/env
SHLVL = 2
```

env 是一个普通的程序，而不是 shell 内置的命令。这里列出设置的值被很多程序所使用，比如，LANG 变量被要显示信息或消息的程序使用，一个浏览器可以用这个变量的值来决定按钮的标识和菜单的选项，DISPLAY 告诉 XWindows 从哪里打开窗口，TERM 告诉 curses(GUI 函数库)使用哪一组屏幕控制代码。

2. 更新环境

(1) var=value

通过对变量赋值就可以更新环境设置。比如，如果浏览器支持法语消息和菜单，可以通过设置 LANG=fr 来启用法语。

(2) export var

使用 shell 内置的命令 export 向环境添加新的变量。如果 var 是一个局部变量，那么它将被添加到环境里。如果 var 不存在，shell 会创建一个。bash 允许通过用 export var=value 将创建和输出合并为一步。

3. 在 C 程序中读入环境

使用标准的 C 库函数 getenv 也可以得到环境变量的值，比如：

```
#include <stdlib.h>
main()
{
    char *cp = getenv("LANG");
    if ( cp != NULL && strcmp(cp , "fr") == 0 )
        printf("Bonjour\n");
    else
```

```
        printf("Hello\n");
}
```

9.6.2　什么是环境以及它是如何工作的

环境是每个程序都可以存取的一个字符串数组，如图 9.6 所示。每个数组中的字符串都以 var=value 这样的形式出现，数组的地址被存放在一个名为 environ 的全局变量里。环境就是 environ 指向的字符串数组，读环境就是读这个字符串数组，改变环境就是改变字符串、改变这个数组中的指针或者将这个全局指针指向其他数组。

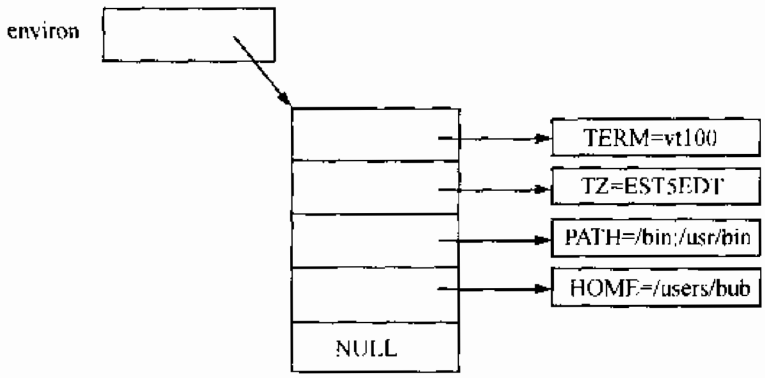


图 9.6　环境是一个指向字符串的指针数组

1. 一个简单的例子

showenv.c 的功能就像命令 env：

```
/* showenv.c - shows how to read and print the environment
 */

extern char ** environ;    /* points to the array of strings */

main()
{
    int i;

    for ( i = 0 ; environ[i] ; i++ )
        printf("%s\n", environ[i] );
}
```

changeenv.c 改变环境，然后运行 env：

```
/* changeenv.c - shows how to change the environment
 *              note: calls "env" to display its new settings
 */
#include<stdio.h>

extern char ** environ;
```

```
main()
{
    char * table[3];

    table[0] = "TERM = vt100";              /* fill the table */
    table[1] = "HOME = /on/the/range";
    table[2] = 0;

    environ = table;                        /* point to that table */

    execlp("env", "env", NULL);             /* exec a program */
}
```

下面是示例：

```
$ ./changeenv
TERM = vt100
HOME = /on/the/range
$
```

仔细看看程序。在程序 changeenv 中创建一个字符串列表，然后调用 execlp 来运行另一个程序 env。第二个程序能够读到这个字符串列表，也就是说通过某些方法，将这个数组从第一个程序空间复制到第二个程序空间了。

2. 但是 exec 清除了所有的数据！

在讨论 exec 系统调用时候知道，对它的调用就像换脑，用目标程序的代码和数据替换调用程序的代码和数据。但是 environ 指针指向的数组是惟一的例外，当内核执行系统调用 execve 时，它将数组和字符串复制到新的程序的数据空间，如图 9.7 所示。

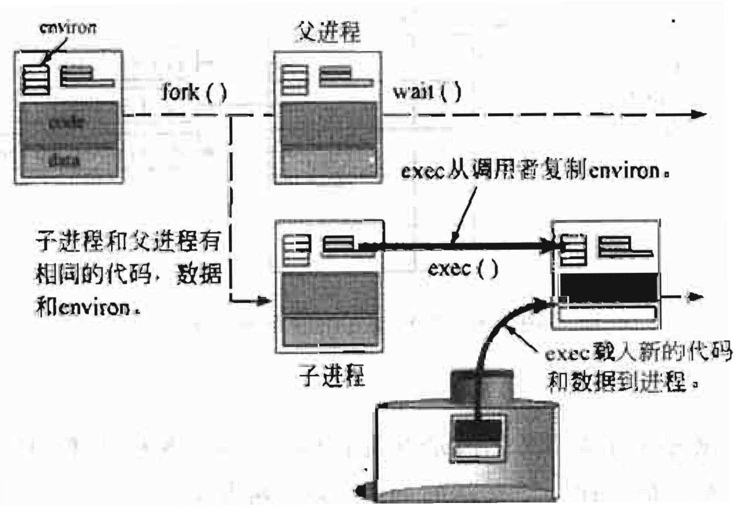


图 9.7 environ 指向的数据在执行 exec()时被复制

在生成子进程的过程中，观察 environ 数组的变化。可以看到，fork 完整地复制父进程，包括代码和数据，数据中包括了环境。exec 清除原来进程中的所有代码和数据，插入新程序

的代码和数据。只有通过参数 execvp 传递的数据和存储在环境中的字符串可以从旧程序复制到新程序。

3. 子进程不能修改父进程的环境

子程序中环境的设置是父进程环境的复本,子进程不能修改父进程的环境。因为在进程调用 fork 和 exec 时整个环境都被自动的复制了,所以通过环境来传递数据比较方便、快捷。

9.6.3 在 smsh 中增加环境处理

现在可以修改 shell 程序使其能够存取环境变量。首先,shell 要将环境中的变量添加到自己的变量列表里。然后,shell 的用户要能够修改和添加环境变量。

1. 存取环境变量

已经知道环境的结构,而且还有一组函数向变量列表添加变量。当 shell 开始运行的时候,环境中的变量将被复制到自己的变量列表里,如图 9.8 所示。一旦这些值被复制到变量列表里,就能用 set 命令和赋值命令来查看和修改这些变量了。

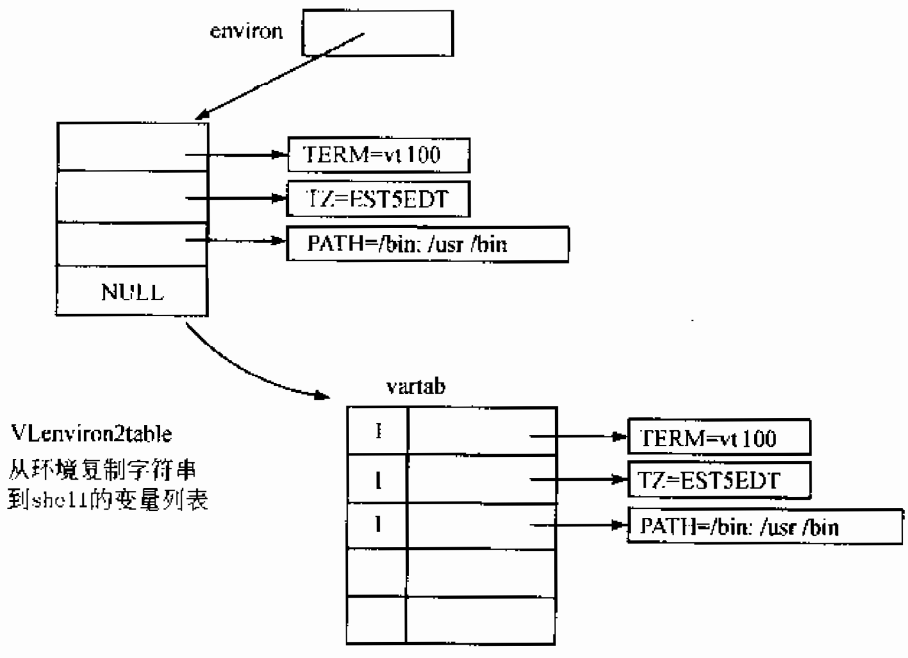


图 9.8 从环境复制值到 vartab

2. 改变环境

对 smsh3 的测试显示了对 TZ 的改动并没有传给 date,现在知道如何修改环境变量了。修改环境最简单的方法是构建一个全新的列表,这个列表包含了 shell 中的所有变量。然后将全局指针 environ 指向这个列表,如图 9.9 所示。调用 exec,内核将这些设置复制到新的程序中。注意,现在没有被引用的环境列表中依旧存储着原来的值。

3. 对 smsh 的修改

在程序流程中添加两步,如图 9.10 所示。这两步通过添加两行代码来实现。

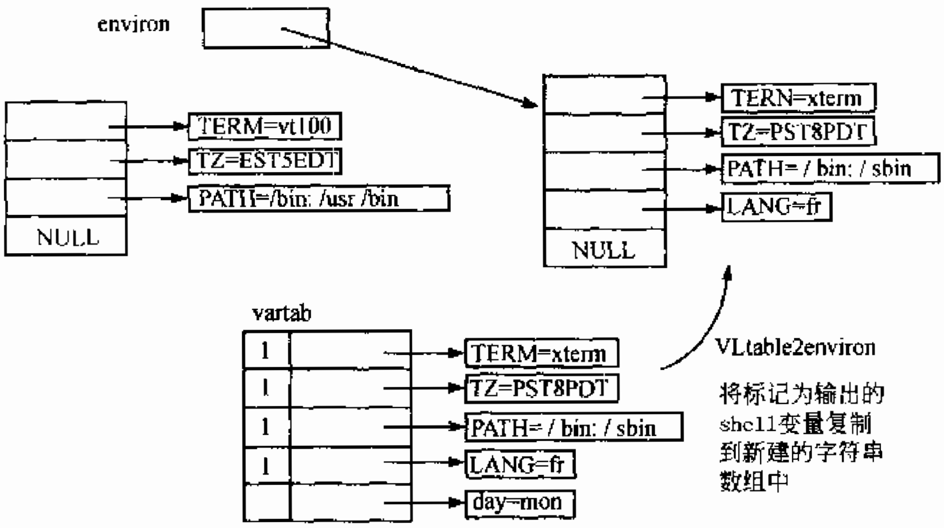


图 9.9 将值从 vartab 复制到新的环境

(1) smsh4.c 中的 setup

```
void setup()
/*
 * purpose: initialize shell
 * returns: nothing. calls fatal() if trouble
 */
{
   extern char ** environ;

   VLenviron2table(environ);
   signal(SIGINT, SIG_IGN);
   signal(SIGQUIT, SIG_IGN);
}
```

(2) execute2.c 中的 execute:

```
if ( (pid = fork()) == -1 )
      perror("fork");
  else if ( pid == 0 ){
      environ = VLtable2environ();/* new line */
      signal(SIGINT, SIG_DFL);
      signal(SIGQUIT, SIG_DFL);
      execvp(argv[0], argv);
      perror("cannot execute command");
      exit(1);
```

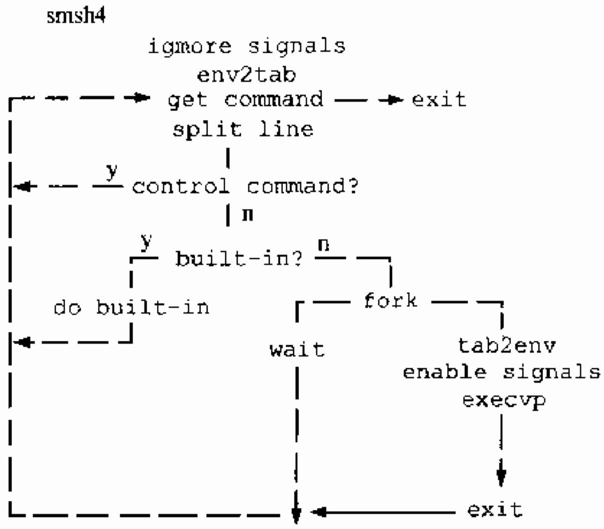


图 9.10　在 smsh 中增加环境处理

4. 测试改动后的程序

```
$ make smsh4
cc  -o smsh4 smsh4.c splitline.c execute2.c process2.c \
   controlflow.c builtin.c varlib.c
$ ./smsh4
>date
Tue Jul 31 09:51:03 EDT 2001
>TZ = PST8PDT
>export TZ
>date
Tue Jul 31 06:51:30 PDT 2001
>
```

用户可以修改和增加环境变量，而且 shell 也会把这些新的值传给它运行的任何程序了。

9.6.4　varlib.c 的代码

```
/* varlib.c
 *
 * a simple storage system to store name = value pairs
 * with facility to mark items as part of the environment
 *
 * interface:
 *     VLstore( name, value )     returns 1 for Ok, 0 for no
 *     VLlookup( name )           returns string or NULL if not there
 *     VLlist()                   prints out current table
 *
```

```
 * environment - related functions
 *    VLexport( name )       adds name to list of env vars
 *    VLtable2environ()      copy from table to environ
 *    VLenviron2table()      copy from environ to table
 *
 * details:
 * the table is stored as an array of structs that
 * contain a flag for global and a single string of
 * the form name = value. This allows EZ addition to the
 * environment. It makes searching pretty easy, as
 * long as you search for "name = "
 *
 */

#include  <stdio.h>
#include  <stdlib.h>
#include  "varlib.h"
#include  <string.h>

#define  MAXVARS  200     /* a linked list would be nicer */

struct var {
      char    * str;      /* name = val string*/
      int     global;     /* a boolean*/
   };

static struct var tab[MAXVARS];          /* the table*/

static char * new_string( char * , char * );/* private methods*/
static struct var * find_item(char * , int);

int VLstore( char * name, char * val )
/*
 * traverse list, if found, replace it, else add at end
 * since there is no delete, a blank one is a free one
 * return 1 if trouble, 0 if ok (like a command)
 */
{
   struct var * itemp;
   char   * s;
   int  rv = 1;

   /* find spot to put it              and make new string */
   if ((itemp = find_item(name,1)) != NULL &&
          (s = new_string(name,val)) != NULL)
   {
      if ( itemp->str )            /* has a val? */
```

```
            free(itemp->str);          /* y: remove it */
        itemp->str = s;
        rv = 0;                        /* ok! */
    }
    return rv;
}

char * new_string( char *name, char *val )
/*
 * returns new string of form name=value or NULL on error
 */
{
    char    *retval;

    retval = malloc( strlen(name) + strlen(val) + 2 );
    if ( retval != NULL )
        sprintf(retval, "%s=%s", name, val );
    return retval;
}

char * VLlookup( char *name )
/*
 * returns value of var or empty string if not there
 */
{
    struct var *itemp;

    if ( (itemp = find_item(name,0)) != NULL )
        return itemp->str + 1 + strlen(name);
    return "";

}

int VLexport( char *name )
/*
 * marks a var for export, adds it if not there
 * returns 1 for no, 0 for ok
 */
{
    struct var *itemp;
    int  rv = 1;

    if ( (itemp = find_item(name,0)) != NULL ){
        itemp->global = 1;
        rv = 0;
    }
    else if ( VLstore(name, "") == 1 )
```

```
        rv = VLexport(name);
    return rv;
}

static struct var * find_item( char * name , int first_blank )
/*
 * searches table for an item
 * returns ptr to struct or NULL if not found
 * OR if (first_blank) then ptr to first blank one
 */
{
    int i;
    int len = strlen(name);
    char   * s;

    for( i = 0 ; i<MAXVARS && tab[i].str != NULL ; i++ )
    {
        s = tab[i].str;
        if ( strncmp(s,name,len) == 0 && s[len] == '=' ){
            return &tab[i];
        }
    }
    if ( i < MAXVARS && first_blank )
        return &tab[i];
    return NULL;
}

void VLlist()
/*
 * performs the shell's set command
 * Lists the contents of the variable table, marking each
 * exported variable with the symbol '*'
 */
{
    int i;
    for(i = 0 ; i<MAXVARS && tab[i].str != NULL ; i++ )
    {
        if ( tab[i].global )
            printf(" *  %s\n", tab[i].str);
        else
            printf(" %s\n", tab[i].str);
        }
    }

    int VLenviron2table(char * env[])
    /*
```

```
 * initialize the variable table by loading array of strings
 * return 1 for ok, 0 for not ok
 */
{
    int     i;
    char    *newstring;

    for(i = 0 ; env[i] != NULL ; i== )
    {
        if ( i == MAXVARS )
            return 0;
        newstring = malloc(1+strlen(env[i]));
        if ( newstring == NULL )
            return 0;
        strcpy(newstring, env[i]);
        tab[i].str = newstring;
        tab[i].global = 1;
    }
    while( i < MAXVARS ){        /* I know we don't need this   */
        tab[i].str = NULL ;      /* static globals are nulled   */
        tab[i==].global = 0;     /* by default                  */
    }
    return 1;
}

char ** VLtable2environ()
/*
 * build an array of pointers suitable for making a new environment
 * note, you need to free() this when done to avoid memory leaks
 */
{
    int     i,              /* index                */
            j,              /* another index        */
            n = 0;          /* counter              */
    char    **envtab;       /* array of pointers    */

    /*
     * first, count the number of global variables
     */

    for( i = 0 ; i<MAXVARS && tab[i].str != NULL ; i++ )
        if ( tab[i].global == 1 )
            n++;

    /* then, allocate space for that many variables   */
    envtab = (char **) malloc( (n+1) * sizeof(char *) );
```

```
    if ( envtab == NULL )
        return NULL;

    /* then, load the array with pointers          */
    for (i = 0, j = 0 ; i<MAXVARS && tab[i].str != NULL ; i++ )
        if ( tab[i].global == 1 )
            envtab[j++] = tab[i].str;
    envtab[j] = NULL;
    return envtab;
}
```

9.7　已实现的 shell 的功能

在本章中，学习了 Unix shell 的可编程特征，还在实现的 shell 中增加了 3 个重要的功能：命令行解析、if..then 语句和变量，使这个小小的 shell 成长迅速。下面是 shell 目前的特征列表：

特征	是否支持	有待改进
命令	运行程序	
变量	=，set	read，$var 替换
if	if..then	else
environ	全部	
exit		exit
cd		cd
>，<，|	不支持	全部

(1) 变量替换

增加变量替换还需进一步研究。在流程中的哪一步将 $ X 替换为 X 的值？注意下面的例子：

```
$ read x
who am i
$ $x
mori.xyz.com! nobody        tty1   Dec 31 13:56
$ grep $x /etc/passwd
grep: am: No such file or directory
grep: i: No such file or directory
$
```

能够从这些输出中得出哪些关于 shell 的分析阶段和变量替换阶段之间关系的信息？这样的设计有什么好处吗？能在这里的程序中添加这一特性吗？

(2) 输入/输出重定向

shell 允许用户将进程的输入和输出重定向到文件或者其他进程。这是如何做到的呢?能够在这里的 shell 中增加这一特征吗? 将在下一章中学习输入/输出重定向。

小　　结

1. 主要内容

- Unix shell 运行一种称为脚本的程序。一个 shell 脚本可以运行程序、接受用户输入、使用变量和使用复杂的控制逻辑。
- if..then 语句依赖于下述惯例:Unix 程序返回 0 以表示成功。shell 使用 wait 来得到程序的退出状态。
- shell 编程语言包括变量。这些变量存储字符串,它们可以在任何命令中使用。shell 变量是脚本的局部变量。
- 每个程序都从调用它的进程中继承一个字符串列表,这个列表被称为环境。环境用来保存会话(session)的全局设置和某个程序的参数设置,shell 允许用户查看和修改环境。

2. 下一步做什么

将学习输入/输出的重定向。

3. 习题

9.1　编写名为 set 的一个 C 程序或脚本,试着用已经实现的 shell 来运行它们。会有什么现象? 写一个名为 no=dice 的 c 程序或者脚本然后试着执行它,又会如何? 用同样的方法试试名为 test 的程序。有没有办法运行这些程序呢?

9.2　能修改 process.c 和 controlflow.c 的设计使之支持嵌套的 if 语句吗? 这就是说,它能处理以下形式的输入吗?

```
if cmd1
then
   if cmd2
   then
      cmd3
   else
      cmd4
   fi
   else
      cmd5
   fi
```

嵌套层次可以是任意深度,而不是像这里所显示的 1 层。需要更多的状态变量吗? 需要一个栈来保存这些状态变量吗? 如果构造一个栈,用递归的方法来解决是不是合理?

9.3　varlib.c 中的函数通过创建一个新的环境数组来更新环境。为什么不用 realloc 来调整原来环境的大小呢？

4. 编程练习

9.4　修改 smsh1.c 使之能够在一行中接受多个命令。要做到这一点最简单的方法是修改 next_cmd 函数，注意不要打印多余的提示符。

9.5　修改 smsh1.c 使之能够接受带可选参数的 exit 命令。保证你的程序拒绝非整数的参数（比如：exit left）。原来的流程中在哪里处理这个命令？需要增加新的节点到原来的流程中去吗？

9.6　修改 process.c 使之能支持 if 控制语句中的 else 部分。

9.7　ok_to_execute 函数使用两个变量来记录当前的区域和状态。可以用一个有多个值的变量来替代原来的两个变量。考虑下面一组状态：

NEUTRAL, IF_SUCCEEDED, IF_FAILED, SKIPPING_THEN, DOING_THEN, SKIPPING_ELSE, DOING_ELSE

修改 controlflow.c 以使用这个单变量的系统。

9.8　修改 smsh1.c 使之能接受 & 命令结束符。以这个符号结束的命令将在后台运行。需要对 next_cmd 做一些修改。

9.9　常规的 shell 直到读到最后 fi（注意 fi 不是 final 的缩写，而是反过来的 if）才执行整个语句块。另一个完全不同的做法是将 if 结构中的所有语句读入一个有三个部分的结构体中。第一个部分是条件命令，第二个部分是 then 区域，最后是 else 区域的命令。

将整个块读入内存后，就能开始执行条件命令，基于它们的结果，执行 then 区域或者 else 区域。写一个采用这个方案的 smsh。你的解应该能接受嵌套的 if 语句。

9.10　在你的 shell 中添加 while 循环。为了增加这个功能，需要把循环体读入内存，小心内存泄漏（memory leak）。

9.11　一个进程有很多属性，其中之一是这个进程的当前目录。Unix 的发明者写了个程序 chdir，还有其他一些标准的目录程序：pwd、ls、mv 等。这些程序已经被废弃，它们的功能直接由 shell 来实现。chdir 应用程序有什么问题吗？在你的 shell 中添加 cd 命令。

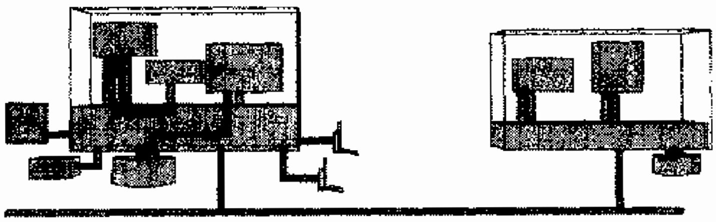
9.12　shell 支持特殊的变量来表示系统设置。比如，变量 $$ 表示 shell 的进程 ID，而 $？表示最后一条命令的退出状态值。在你的程序中增加这些变量。

9.13　在标准 Unix shell 的命令行中，可以将引号引起来的部分作为一个独立的参数。一个命令像 vi "My Book Report"包含两个命令行参数。使你的 shell 接受引号。

在 shell 的哪一部分处理引号？考虑命令 rm "file1. c;2"。就算你的程序将分号理解为命令分隔符，这个表达式还是应该被理解为一条有两个参数的命令。

9.14 很多 shell 允许用户通过对一个特定的变量赋值来设置命令提示符，在你的 shell 中增加这个特征。在自己的 shell 中定义一个变量来表示提示符。sh 和 bash 用变量 PS1，而 csh 家族用变量 prompt。

第 10 章　I/O 重定向和管道



概念与技巧

- I/O 重定向：概念与原因
- 标准输入、输出和标准错误的定义
- 重定向标准 I/O 到文件
- 使用 fork 来为其他程序重定向
- 管道(Pipe)
- 创建管道后调用 fork

相关的系统调用与函数

- dup、dup2
- pipe

10.1　shell 编程

下面这组命令是如何工作的？

```
ls > my.files
who | sort>userlist
```

shell 是如何告诉程序将结果输出到文件而不是屏幕的呢？shell 又是如何将一个进程的输出流连接到另一个进程的输入流的呢？标准输入(standard input)这个术语是什么意思？

本章将关注进程间通信的一种特殊形式：输入/输出重定向和管道(I/O redirection and pipes)。首先将介绍在编写 shell 脚本时 I/O 重定向和管道所起的作用。然后，本章将介绍操作系统中对 I/O 重定向的支持。最后，写一个程序来改变进程的输入和输出流。

10.2　一个 shell 应用程序：监视系统用户

考虑一下这个问题：你的许多朋友和你使用同一个 Unix 系统。你希望编写一个程序，当其他用户登录系统或注销时通知你。这样你就可以了解朋友们的活动。

可以写一个使用 utmp 文件和间隔计数器的 C 程序来完成任务。程序打开 utmp 文件，记录下用户列表，休眠一段时间后再重新扫描此文件，并将变化报告出来。

一个更简单的办法就是写一个 shell 脚本。Unix 中有一个列出当前用户的命令：who。Unix 中同样包含了休眠和处理字符串列表的程序。下面是一个 Unix 的脚本，用来报告所有的登录和注销情况。

```
Logic                                          shell code
---------------------------                    --------------------
get list of users(call it prev)                who | sort > prev
while true                                     while true ; do
  sleep                                          sleep 60
  get list of users(call it curr)                who | sort > curr
  compare lists                                  echo "logged out: "
    in prev, not in curr -> logout               comm -23 prev curr
                                                 echo "logged in: "
    in curr, not in prev -> login                comm -13 prev curr
  make prev = curr                               mv curr prev
repeat                                         done
```

此脚本使用了 Unix 系统所提供的 7 个工具、一个 while 循环和 I/O 重定向，编写这个程序解决了问题。仔细看一下这些程序的细节，以及它们之间的连接。

脚本中的第一行建立了一个在此脚本运行时已登录用户的列表，并按用户名进行排序。who 命令输出用户列表，而 sort 命令将列表作为输入读进，然后输出一个排好序的列表。

命令 who | sort > prev 告诉 shell 同时执行 who 和 sort，将 who 的输出直接送到 sort 的输入，如图 10.1 所示。who 命令并不一定要在 sort 命令开始读取和排序之前完成对 utmp 文件的分析。这两个进程以很小的时间间隔为单位来调度，它们和系统中的其他进程一起分享 CPU 时间。然后，sort > prev 告诉 shell 将 sort 的输出送至 prev 文件中。若此文件不存在，则创建此文件；若已经存在，则替换其内容。

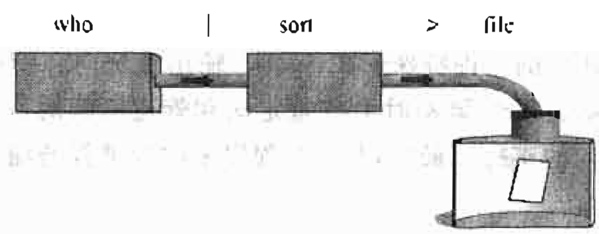


图 10.1　将 who 的输出连结到 sort 的输入

在休眠一分钟之后，脚本在文件 curr 中创建了一个新的用户列表。如何来比较两个排好序的登录记录列表呢？使用 Unix 的工具 comm(如图 10.2 所示)，可以找出两个文件中共有的行。比较两个文件可以得到三个子集：仅文件 1 有的行，仅文件 2 有的行，两者共有的行。comm 命令比较了两个排过序的列表，并将此三列打印出来，这里的每一列代表一个子集的内容。可以使用命令行选项来让结果只出现其中的任意一列或两列。比如说，如下两个命令：

```
comm -23 prev curr # 删除第二列和第三列 =>仅显示 prev 中的内容
comm -13 prev curr # 删除第一列和第三列 =>仅显示 curr 中的内容
```

生成所需要的两个集合：前一个列表中有而当前列表中没有的登录记录(注销的用户)，以及当前列表中有而前一个列表中没有的登录记录(新登录用户)。

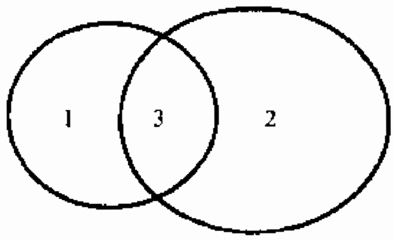


图 10.2 comm 比较两个列表，输出三个集合

最后，命令 mv curr prev 将当前列表文件 curr 更名为 prev，并替换原来的 prev 文件。

watch.sh 脚本体现了三个重要的思路：

(1) shell 脚本的功能——与 C 相比简单易用；

(2) 软件工具的灵活性——每一个工具完成一项特定的、通用的功能；

(3) I/O 重定向和管道的使用和作用。

程序 watch.sh 展示了如何使用“>”操作符来把文件看成任意大小和结构的变量。类似于某人用 C 写了如下的调用：

```
x = func_a(func_b(y));   /* 将 func_b 的结果作为 func_a 的输入*/
```

用 shell 写，就是：

```
prog_b | prog_a > x # 将 prog_b 的结果作为 prog_a 的输入并将最终结果放入 x
```

这些程序如何工作的？shell 在进程的连接中起什么样的作用呢？内核起什么作用？单个程序又起什么作用？

10.3 标准 I/O 与重定向的若干概念

所有的 Unix I/O 重定向都基于标准数据流的原理。考虑一下 sort 工具是如何工作的。sort 从一个数据流中读取字节，再将结果输出到另一个流中，同时若有错误发生，则将错误报

告给第三个流。如果忽略这些标准流的去向问题，sort 工具的基本原型就如图 10.3 所示。三个数据流分别如下：

- 标准输入——需要处理的数据流
- 标准输出——结果数据流
- 标准错误输出——错误消息流

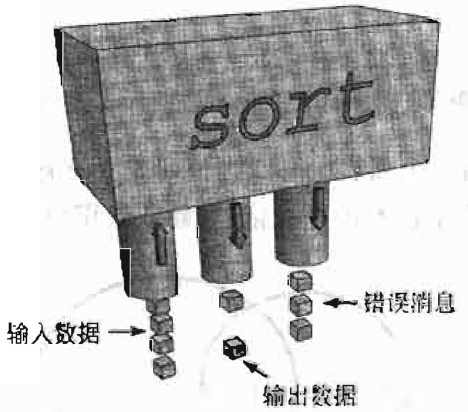


图 10.3　sort 工具将输入读进并输出结果和错误消息

10.3.1　概念 1：3 个标准文件描述符

所有的 Unix 工具都使用图 10.3 中所示的三种流的模型。此模型通过一个简单的规则来实现。这三种流的每一种都是一个特别的文件描述符，其细节如图 10.4 所示。

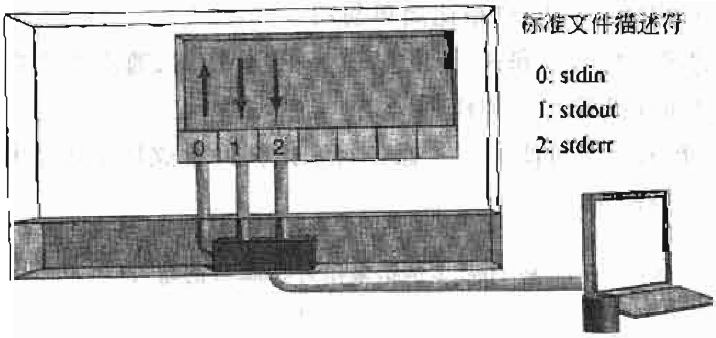


图 10.4　3 个特殊的文件描述符

概念：所有的 Unix 工具都使用文件描述符 0、1 和 2。

标准输入文件的描述符是 0，标准输出的文件描述符是 1，而标准错误输出的文件描述符则是 2。Unix 假设文件描述符 0、1、2 已经被打开，可以分别进行读、写和写的操作了。

10.3.2　默认的连接：tty

通常通过 shell 命令行运行 Unix 系统工具时，stdin、stdout 和 stderr 连接在终端上。因此，工具从键盘读取数据并且把输出和错误消息写到屏幕。举例来说，如果输入 sort 并按下回车键，终端将会被连接到 sort 工具上。随便输入几行文字，当按 Ctrl－D 键来结束文字输

入的时候，sort 程序对输入进行排序并将结果写到 stdout。

大部分的 Unix 工具处理从文件或标准输入读入的数据。如果在命令行上给出了文件名，工具将从文件读取数据。若无文件名，程序则从标准输入读取数据。

10.3.3　程序都输出到 stdout

从另一方面说，大多数程序并不接收输出文件名；它们总是将结果写到文件描述符 1，并将错误消息写到文件描述符 2①。如果希望将进程的输出写到文件或另一个进程的输入去，就必须重定向相应的文件描述符。

10.3.4　重定向 I/O 的是 shell 而不是程序

通过使用输出重定向标志，命令 cmd>filename 告诉 shell 将文件描述符 1 定位到文件。于是 shell 就将文件描述符与指定的文件连接起来。

程序则持续不断地将数据写到文件描述符 1 中，根本没有意识到数据的目的地已经改变了。下面的程序 listargs.c 展示了程序甚至没有看到命令行中的重定向符号：

```
/* listargs.c
 *           print the number of command line args, list the args,
 *       then print a message to stderr
 */
#include <stdio.h>
main( int ac, char * av[] )
{
    int i;

    printf("Number of args: %d, Args are: \n", ac);
    for(i = 0; i<ac; i++ )
        printf("args[%d] %s\n", i, av[i]);

     fprintf(stderr,"This message is sent to stderr.\n");
}
```

程序 listargs 将命令行参数打印到标准输出。注意 listargs 并没有打印出重定向符号和文件名。

```
$ cc listargs.c -o listargs
$ ./listargs testing one two
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
This message is sent to stderr.
```

① sort 和 dd 命令允许覆盖 stdout，但这是由于其他的原因。

```
$ ./listargs testing one two > xyz
This message is sent to stder.
$ cat xyz
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
$ ./listargs testing >xyz one two 2> oops
$ cat xyz
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
$ cat oops
This message is sent to stderr.
```

这些例子验证了关于 shell 输出重定向的一些重要概念。最重要的一点是 shell 并不将重定向标记和文件名传递给程序。

第二个概念是重定向可以出现在命令行中的任何地方，并且在重定向标识符周围并不需要空格来区分。甚至一个像 >listing ls 这样的命令也是可以接受的。这样，">"符号并不能终止命令和参数，它只不过是一个附加的请求而已。

最后一个概念是许多版本的 shell 都提供对重定向其他文件描述符的支持。例如，2>filename 即重定向文件描述符 2，也就是将标准错误输出到给定的文件中。

10.3.5 理解 I/O 重定向

在 watch.sh 中可以看到，I/O 重定向是 Unix 程序设计中一个重要部分。同样在 listargs.c 中看到，是 shell，而非程序将输入和输出重定向的。

但 shell 是如何重定向 I/O 的呢？怎样写重定向 I/O 的程序呢？本章的工作就是编写可以完成三个基本的重定向操作的程序：

- **who>userlist**　　将 stdout 连接到一个文件
- **sort<data**　　将 stdin 连接到一个文件
- who|sort　　将 stdout 连接到 stdin

10.3.6 概念 2："最低可用文件描述符(Lowest-Available-fd)"原则

那么什么是文件描述符呢？文件描述符的概念非常简单：它是一个数组的索引号。每个进程都有其打开的一组文件。这些打开的文件被保持在一个数组中。文件描述符即为某文件在此数组中的索引。图 10.5 显示了"最低可用文件描述符(Lowest-Available-fd)"原则。

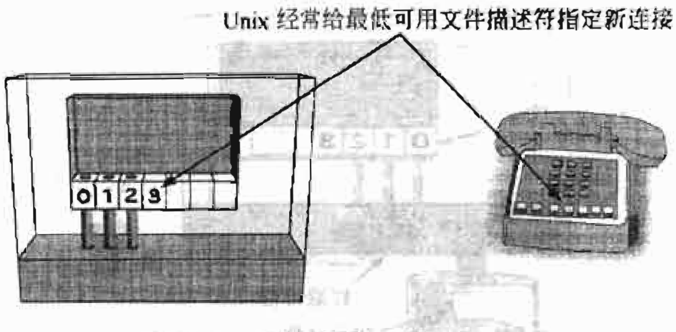


图 10.5　最低可用文件描述符原则

概念：当打开文件时，为此文件安排的描述符总是此数组中最低可用位置的索引。

通过文件描述符建立一个新的连接就像在一条多路电话上接收一个连接一样。每当有用户拨一个电话号码，内部电话系统为这个拨号请求分配一条内部的线路号。在许多这样的系统上，下一个打进来的电话就被分配给最小可用的线路号。

10.3.7　两个概念的结合

已经介绍了两个基本的概念。首先，Unix 进程使用文件描述符 0、1、2 作为标准输入、输出和错误的通道。其次，当进程请求一个新的文件描述符的时候，系统内核将最低可用的文件描述符赋给它。将这两个概念结合在一起，大家就可以理解 I/O 重定向是如何工作的了，也就可以自己写出程序来完成 I/O 的重定向。

10.4　如何将 stdin 定向到文件

下面将详细地考察，程序如何将标准输入重定向以至可以从文件中读取数据。更加精确一点说，进程并不是从文件读数据，而是从文件描述符读数据。如果将文件描述符 0 定位到一个文件，那么此文件就成为标准输入的源。

下面将考察三种将标准输入定位到文件的方法。其中有些方法并不适合于文件，但使用管道的时候，这些方法都是必要的。

10.4.1　方法 1：close then open

第一种方法是 close-then-open 策略。这种技术类似于挂断电话释放一条线路，然后再将电话拎起从而得到另一条线路。具体步骤如下。

开始的时候，系统中采用的是典型的设置。即三种标准流是被连接到终端设备上的。输入的数据流经过文件描述符 0 而输出的流经过文件描述符 1 和 2，如见图 10.6 所示。

接下来，第一步是 close(0)，即将标准输入的连接挂断。这里调用 close(0)将标准输入与终端设备的连接切断。图 10.7 中显示了当前文件描述符数组中的第一个元素现在处在空闲状态。

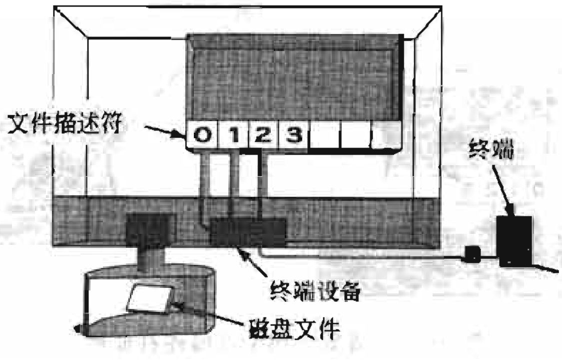


图 10.6　典型的初始化配置

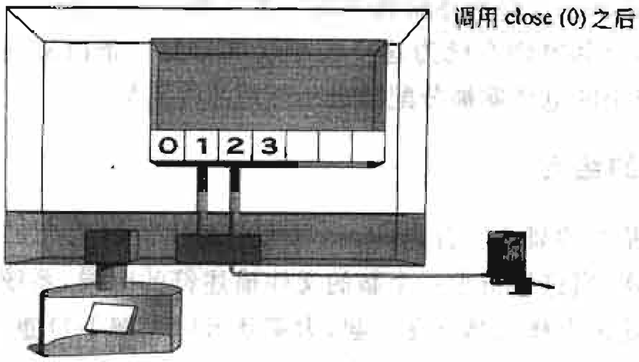


图 10.7　stdin 被关闭

最后，使用 open(filename,O_RDONLY)打开一个想连接到 stdin 上的文件。当前的最低可用文件描述符是 0，因此所打开的文件将被连接到标准输入上去。如图 10.8 所示，任何从标准输入读取数据的函数都将从此文件中读入。



图 10.8　stdin 现在已经连接到文件上了

下面的程序即使用 close-then-open 方法：

```
*/ stdinredir1.c
* purpose: show how to redirect standard input by replacing file
*       descriptor 0 with a connection to a file.
* action: reads three lines from standard input, then
*       closes fd 0, opens a disk file, then reads in
```

```
 *      three more lines from standard input
 */
#include    <stdio.h>
#include    <fcntl.h>

main()
{
    int    fd ;
    char   line[100];

    /* read and print three lines */

    fgets( line, 100, stdin ); printf( "%s", line );
    fgets( line, 100, stdin ); printf( "%s", line );
    fgets( line, 100, stdin ); printf( "%s", line );

    /* redirect input */

    close(0);
    fd = open("/etc/passwd", O_RDONLY);
    if ( fd != 0 ){
        fprintf(stderr,"Could not open data as fd 0\n");
        exit(1);
    }

    /* read and print three lines */

    fgets( line, 100, stdin ); printf( "%s", line );
    fgets( line, 100, stdin ); printf( "%s", line );
    fgets( line, 100, stdin ); printf( "%s", line );
}
```

程序 stdinreader1 从标准输入读取并打印了三行字符串，然后重定向标准输入，之后又从标准输入中读取并打印了三行字符串。stdinreader1 从键盘读取了前三行字符串，而后三行字符串则是从 passwd 文件中读出的：

```
$ ./stdinredir1
line1
line1
testing line2
testing line2
line 3 here
line 3 here
root: x: 0: 0: root: /root: /bin/bash
bin: x: 1: 1: bin: /bin:
daemon: x: 2: 2: daemon: /sbin:
$
```

此程序并没有什么特别的地方，它仅仅挂断电话又拨了一个新的号码而已。当连接建立起来后，就可以从标准输入的一个新的源接收数据了。

10.4.2　方法 2：open..close..dup..close

考虑一下这种情况：电话响了，你拿起了楼上的分机，但你意识到自己应该下楼去接电话。于是你让楼下的人把电话拎起，这样就有两个连接，然后把楼上的分机挂断，此时楼下的电话是惟一的连接了。这种情况大家是不是很熟悉？其实这种方法的思路就是从楼上的电话复制一个连接到楼下，然后就可以在不断线的情况下将楼上的连接切断。

如图 10.9 所示，Unix 系统调用 dup 建立指向已经存在的文件描述符的第二个连接。这种方法需要 4 个步骤。

(1) open(file)

第一步是打开 stdin 将要重定向的文件。这个调用返回一个文件描述符，这个描述符并不是 0，因为 0 在当前已经被打开了。

(2) close(0)

下一步是将文件描述符 0 关闭。文件描述符 0 现在已经空闲了。

(3) dup(fd)

系统调用 dup(fd)将文件描述符 fd 做了一个复制。此次复制使用最低可用文件描述符号。因此，获得的文件描述符是 0。这样，就将磁盘文件与文件描述符 0 连接在一起了。

(4) close(fd)

最后，使用 close(fd)来关闭文件的原始连接，只留下文件描述符 0 的连接。将这种方法与把电话从一个分机转移到另一个分机的技术做一个比较。

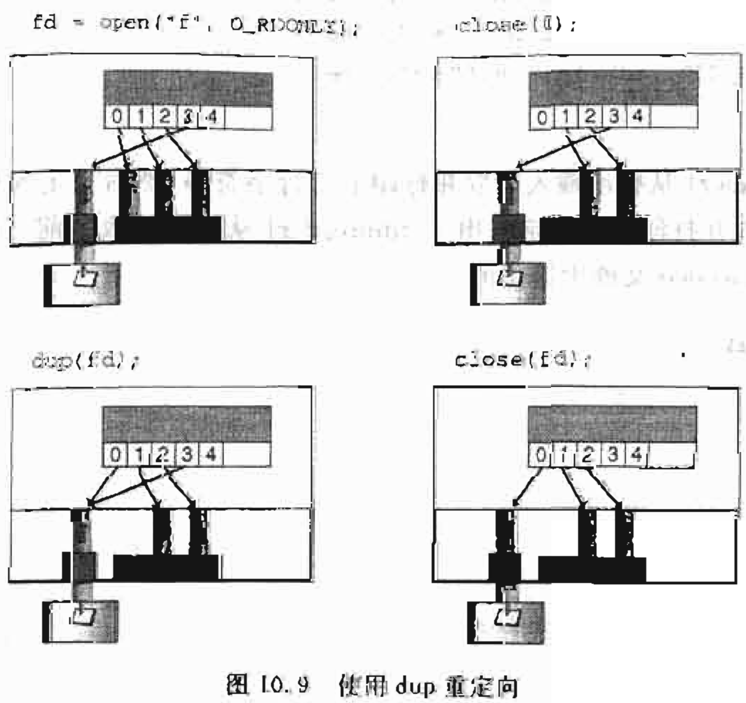


图 10.9　使用 dup 重定向

下面的程序 stdinredir2.c 使用了第二种方法：

```
/* stdinredir2.c
 * shows two more methods for redirecting standard input
 * use #define to set one or the other
 */
#include    <stdio.h>
#include    <fcntl.h>

/* #define   CLOSE_DUP     /* open, close, dup, close */
/* #define   USE_DUP2      /* open, dup2, close */

main()
{
    int     fd ;
    int     newfd;
    char    line[100];

    /* read and print three lines */

    fgets( line, 100, stdin ); printf( "%s", line );
    fgets( line, 100, stdin ); printf( "%s", line );
    fgets( line, 100, stdin ); printf( "%s", line );

    /* redirect input */

    fd = open("data", O_RDONLY);        /* open the disk file */
#ifdef CLOSE_DUP
    close(0);
    newfd = dup(fd);                    /* copy open fd to 0 */
#else
    newfd = dup2(fd,0);                 /* close 0, dup fd to 0 */
#endif
    if ( newfd != 0 ){
        fprintf(stderr,"Could not duplicate fd to 0\n");
        exit(1);
    }
    close(fd);                          /* close original fd */

    /* read and print three lines */

    fgets( line, 100, stdin ); printf( "%s", line );
    fgets( line, 100, stdin ); printf( "%s", line );
    fgets( line, 100, stdin ); printf( "%s", line );
}
```

介绍上面提到的这个包含有 4 个步骤的方法的主要目的是为了让大家了解 dup 系统调用，这个调用在后面学习管道的时候是非常重要的。一个简单一点的方案是将 close(0) 和

dup(fd)结合在一起作为一个单独的系统调用 dup2。

10.4.3　系统调用 dup 小结

dup, dup2	
目标	复制一个文件描述符
头文件	#include <unistd.h>
函数原型	newfd = dup(oldfd); newfd = dup2(oldfd,newfd);
参数	oldfd 需要复制的文件描述符 newfd 复制 oldfd 后得到的文件描述符
返回值	-1　　发生错误 newfd　　新的文件描述符

系统调用 dup 复制了文件描述符 oldfd。而 dup2 将 oldfd 文件描述符复制给 newfd。两个文件描述符都指向同一个打开的文件。这两个调用都返回新的文件描述符,若发生错误,则返回-1。

10.4.4　方法 3: open..dup2..close

程序 stdinredir2.c 包含了条件编译代码 #ifdef,用系统调用 dup2(fd,0)来替换 close(0)和 dup(fd)。dup2(orig,new)将文件描述符 old 复制到文件描述符 new,在此之前它先将文件描述符 new 上已经存在的连接关闭。

10.4.5　shell 为其他程序重定向 stdin

这些例子显示了程序如何将标准输入重定向到文件。实际上,如果程序希望读取文件,它直接打开文件就可以了,根本不需要将标准输入重定向到文件。这些例子的真正意义在于说明一个程序如何将标准输入重定向到别的程序。

10.5　为其他程序重定向 I/O: who > userlist

当某用户输入 who>userlist,shell 运行 who 程序,并将 who 的标准输出重定向到名为 userlist 的文件上。这是如何完成的呢?

关键之处就在于 fork 和 exec 之间的时间间隙。在 fork 执行之后,子进程仍然在运行 shell 程序,并准备执行 exec。exec 将替换进程中运行的程序,但它不会改变进程的属性和进程中所有的连接。也就是说,在运行过 exec 之后,进程的用户 ID 不会改变,其优先级不会改变,并且其文件描述符也和运行 exec 之前一样。注意,程序得到的是载入它的进程所打开的文件。图 10.10 展示了子进程的输出重定向。

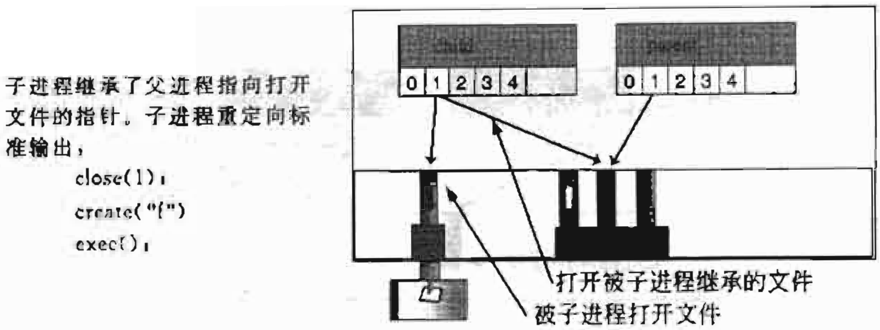


图 10.10　shell 为子进程重定向其输出

看一下如何使用这个原则来重定向标准输出。

1. *初始情况*

如图 10.11 所示，进程运行在用户空间中。文件描述符 1 连接在打开的文件 f 上。为了使这幅图清楚易理解，其他打开的文件并未画出来。

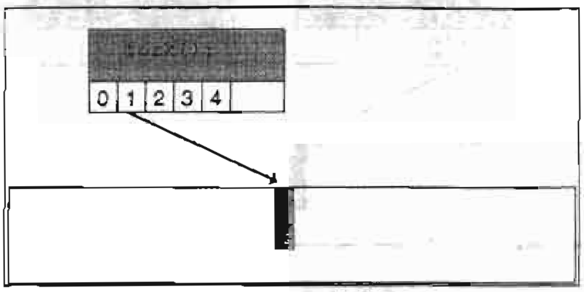


图 10.11　在调用 fork 之间的进程以及它的标准输出

2. *父进程调用 fork 之后*

如图 10.12 所示，新的进程出现了。此进程与原始进程运行相同的代码，但它知道自己是子进程。此进程包含了与父进程相同的代码、数据和打开文件的文件描述符。因此文件描述符 1 依然指向的是文件 f。然后子进程调用了 close(1)。

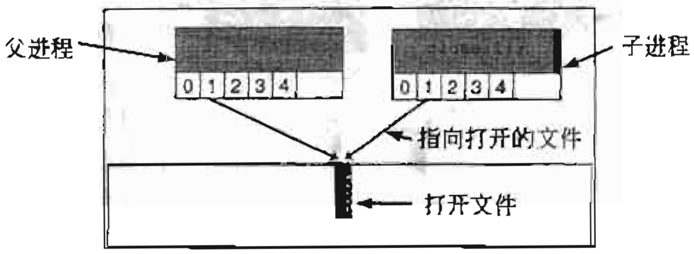


图 10.12　子进程的标准输出从父进程那儿继承而得

3. *在子进程调用 close(1)之后*

如图 10.13 所示，父进程并没有调用 close(1)，因此父进程中的文件描述符 1 仍然指向 f。子进程调用 close(1)之后，文件描述符 1 变成了最低未用文件描述符。子进程现在试着

打开文件 g。

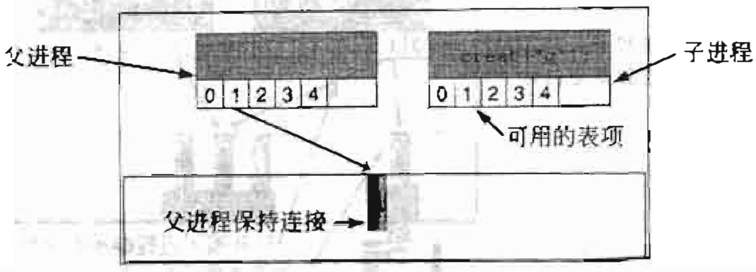


图 10.13　子进程可以关闭其标准输出

4. *在子进程调用 creat("g",m)之后*

如图 10.14 所示,文件描述符 1 被连接到文件 g。子进程的标准输出被重定向到 g。子进程然后调用 exec 来运行 who。

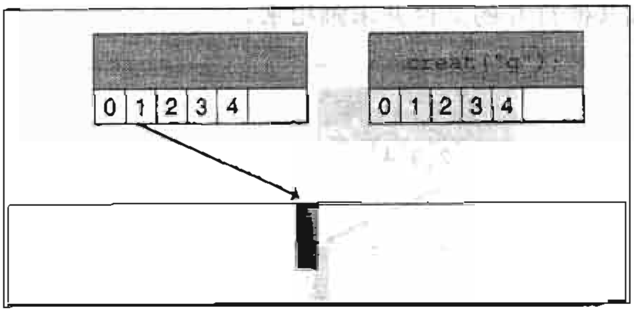


图 10.14　子进程打开一个新的文件得到 fd=1

5. *在子进程使用 exec 执行新程序之后*

如图 10.15 所示,子进程执行了 who 程序。于是子进程中的代码和数据都被 who 程序的代码和数据所替代了,然而文件描述符被保留下来。打开的文件并非是程序的代码也不是数据,它们属于进程的属性,因此 exec 调用并不改变它们。

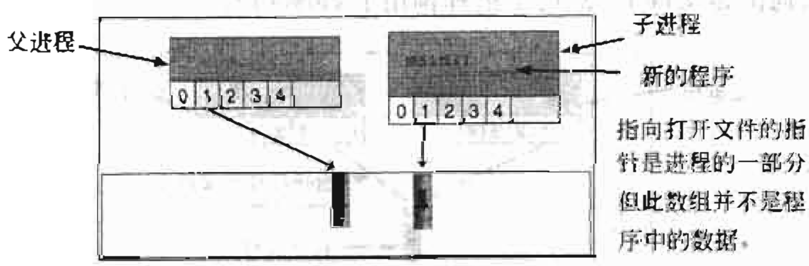


图 10.15　子进程运行程序并将标准输出重定向

who 命令将当前用户列表送至文件描述符 1。其实这组字节已经被写到文件 g 中去了,而 who 命令却毫不知晓。

下面的程序 whotofile.c 展示了上面所说的这种方法:

```
/* whotofile.c
 * purpose: show how to redirect output for another program
 *    idea: fork, then in the child, redirect output, then exec
 */
#include    <stdio.h>

main()
{
    int    pid;
    int    fd;
    printf("About to run who into a file\n");

    /* create a new process or quit */
    if( (pid = fork() ) == -1 ){
        perror("fork"); exit(1);
    }
    /* child does the work */
    if ( pid == 0 ){
        close(1);                             /* close, */
        fd = creat( "userlist", 0644 );       /* then open */
        execlp( "who", "who", NULL );         /* and run */
        perror("execlp");
        exit(1);
    }
    /* parent waits then reports */
    if ( pid != 0 ){
        wait(NULL);
        printf("Done running who. results in userlist\n");
    }
}
```

重定向到文件的小结

共有三个基本的概念，利用它们使得 Unix 下的程序可以轻易地将标准输入、输出和错误信息输出连接到文件：

(1) 标准输入、输出以及错误输出分别对应于文件描述符 0、1、2；

(2) 内核总是使用最低可用文件描述符；

(3) 文件描述符集合通过 exec 调用传递，且不会被改变。

shell 使用进程通过 fork 产生子进程与子进程调用 exec 之间的时间间隔来重定向标准输入、输出到文件。

shell 同样支持下面几种形式的命令：

```
who >> userlog
sort < data
```

编写可以支持以上两种操作的代码就留给大家作为练习去完成。

10.6 管道编程

现在已经学习了如何编写程序将标准输出重定向到文件。下面将要讨论如何使用管道来连接一个进程的输出和另一个进程的输入。图 10.16 展示了管道的工作原理。管道是内核中的一个单向的数据通道。管道有一个读取端和一个写入端。实现 who|sort 这样的操作,需要两种技巧:如何创建管道,以及如何将标准输入和输出通过管道连接起来。

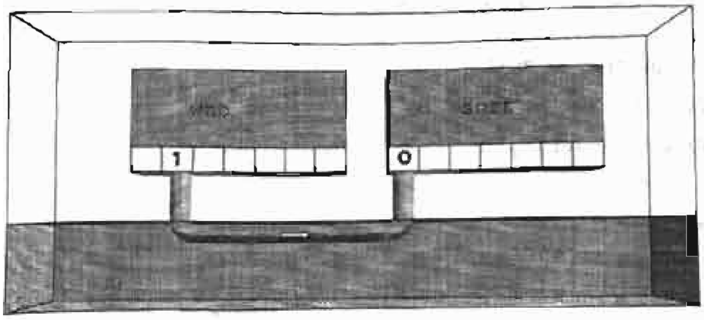


图 10.16　两个进程由管道连接在一起

10.6.1 创建管道

图 10.17 所示即为一根管道。可以使用如下的系统调用来创建管道。

	pipe	
目标	创建管道	
头文件	#include <unistd.h>	
函数原型	result = pipe(int array[2]);	
参数	array	包含两个 int 类型数据的数组
返回值	−1	发生错误
	0	成功

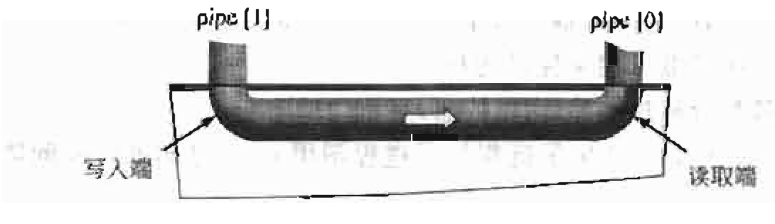


图 10.17　管道

调用 pipe 来创建管道并将其两端连接到两个文件描述符。array[0]为读数据端的文件

描述符，而 array[1]则为写数据端的文件描述符。像一个打开的文件的内部情况一样，管道的内部实现隐藏在内核中，进程只能看见两个文件描述符。

图 10.18 显示了进程创建一个管道前后的状况。前一张图（调用 pipe 之前）显示了标准文件描述符集。后一张图（调用 pipe 之后）显示了内核中新创建的管道，以及进程到管道的两个连接。注意，类似于 open 调用，pipe 调用也使用最低可用文件描述符。

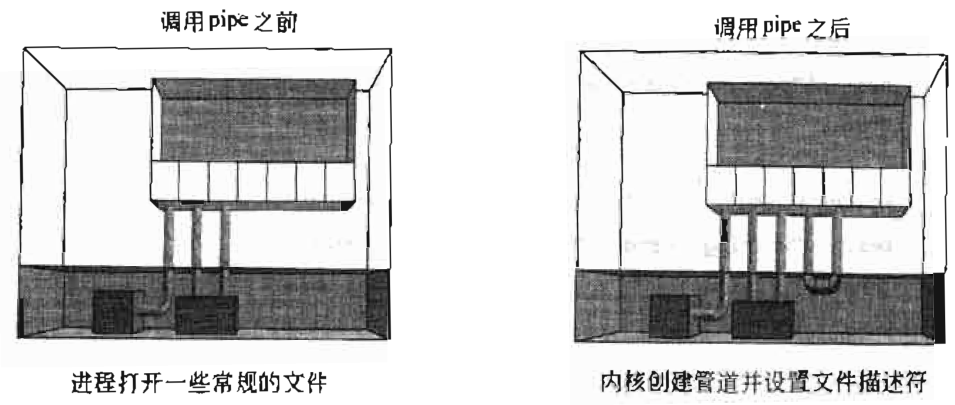


图 10.18 进程创建管道

下面的程序 pipedemo.c 展示了如何创建管道并使用管道来向自己发送数据：

```
/* pipedemo.c    * Demonstrates: how to create and use a pipe
 *               * Effect: creates a pipe, writes into writing
 *                 end, then runs around and reads from reading
 *                 end. A little weird, but demonstrates the idea.
 */
#include <stdio.h>
#include <unistd.h>

main()
{
    int     len, i, apipe [2];     /* two file descriptors    */
    char    buf [BUFSIZ];          /* for reading end         */

    /* get a pipe */
    if ( pipe ( apipe ) == -1 ){
        perror("could not make pipe");
        exit(1);
    }
    printf("Got a pipe! It is file descriptors: { %d %d }\n",
                    apipe[0], apipe[1]);

    /* read from stdin, write into pipe, read from pipe, print */

    while ( fgets(buf, BUFSIZ, stdin) ){
        len = strlen( buf );
```

```
            if ( write( apipe[1], buf, len) != len ){         /* send */
                perror("writing to pipe");                     /* down */
                break;                                         /* pipe */
            }
            for (i = 0 ; i<len ; i++ )                         /* wipe */
                buf[i] = 'X' ;
            len  = read( apipe[0], buf, BUFSIZ ) ;            /* read */
            if ( len == -1 ){                                  /* from */
                perror("reading from pipe");                   /* pipe */
                break;                                         /* pipe */
            }
            if ( write( 1 , buf, len ) != len ){               /* send */
                perror("writing to stdout");                   /* to */
                break;                                         /* pipe */
            }
        }
}
```

图 10.19 显示了从键盘到进程，从进程到管道、再从管道到进程以及从进程回到终端的数据传输流。

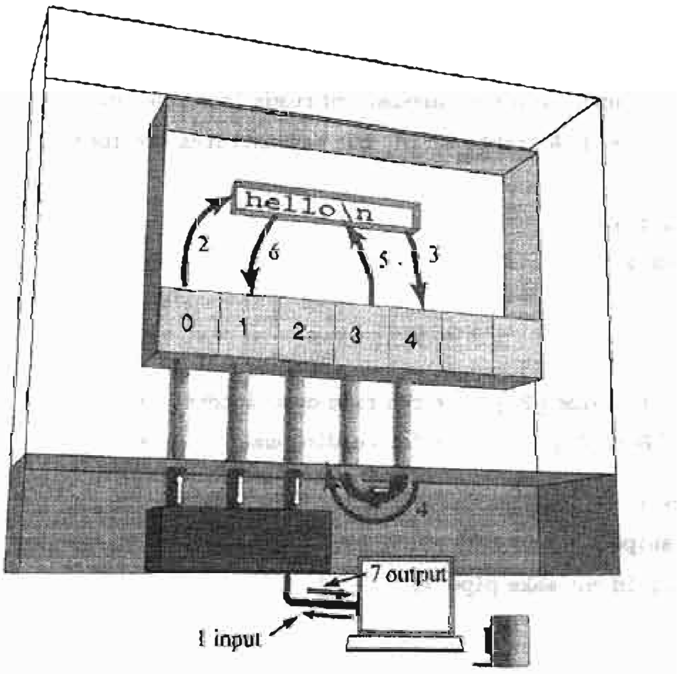


图 10.19 pipedemo.c 中的数据流

现在已经学习了如何创建管道，如何向管道中写数据以及如何从管道中读取数据。实际上，很少会有程序用管道向自己发送数据。将 pipe 和 fork 结合起来，就可以连接两个不同的进程了。

10.6.2 使用 fork 来共享管道

当进程创建一个管道之后，该进程就有了连向管道两端的连接。当这个进程调用 fork 的时候，它的子进程也得到了这两个连向管道的连接，如图 10.20 所示。父进程和子进程都可以将数据写到管道的写数据端口，并从读数据端口将数据读出，如图 10.21 所示。两个进程都可以读写管道，但是当一个进程读、另一个进程写的时候，管道的使用效率是最高的。

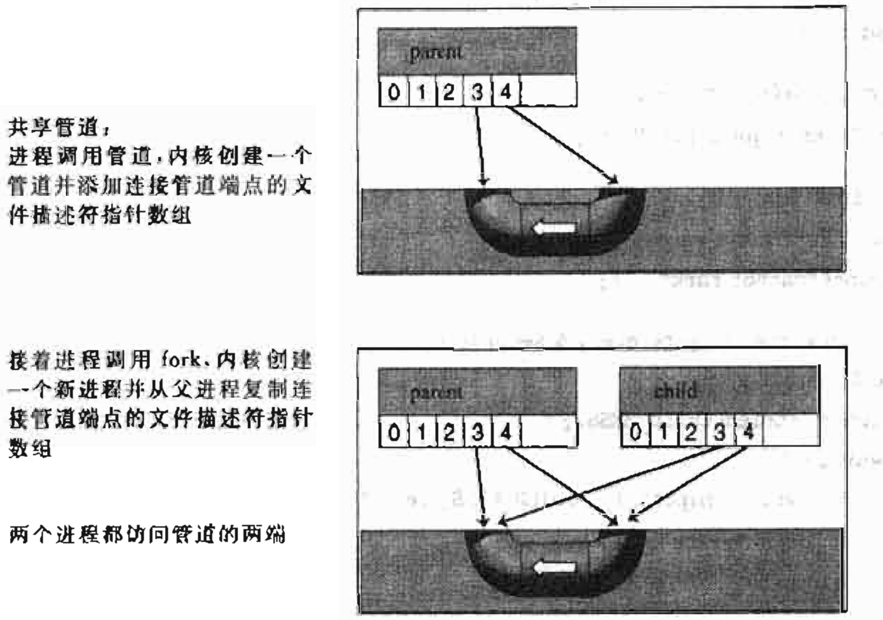


图 10.20 共享管道

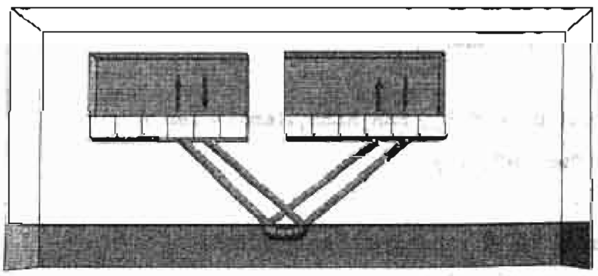


图 10.21 进程之间的数据流

下面的程序 pipedemo2.c 说明了如何将 pipe 和 fork 结合起来，创建一对通过管道来通信的进程。

```
/* pipedemo2.c    * Demonstrates how pipe is duplicated in fork()
 *                * Parent continues to write and read pipe,
 *                  but child also writes to the pipe
 */
#include  <stdio.h>

#define    CHILD_MESS     "I want a cookie\n"
```

```
#define    PAR_MESS        "testing..\n"
#define    oops(m,x)       { perror(m); exit(x); }

main()
{
    int     pipefd[2];      /* the pipe     */
    int     len;            /* for write    */
    char    buf[BUFSIZ];    /* for read     */
    int read_len;

    if ( pipe( pipefd ) == -1 )
        oops("cannot get a pipe", 1);

    switch( fork() ){
        case -1:
            oops("cannot fork", 2);

        /* child writes to pipe every 5 seconds */
        case 0:
            len = strlen(CHILD_MESS);
            while ( 1 ){
                if (write( pipefd[1], CHILD_MESS, len) != len )
                    oops("write", 3);
                sleep(5);
            }

        /* parent reads from pipe and also writes to pipe */
        default:
            len = strlen( PAR_MESS );
            while ( 1 ){
                if ( write( pipefd[1], PAR_MESS, len)!= len )
                    oops("write", 4);
                sleep(1);
                read_len = read( pipefd[0], buf, BUFSIZ );
                if ( read_len <= 0 )
                    break;
                write( 1, buf, read_len );
            }
    }
}
```

10.6.3 使用 pipe、fork 以及 exec

本章已经介绍了各种技巧和思路来编写将 who 的输出连接到 sort 的输入的程序。大家应该已经了解了如何去创建管道，如何在进程间共享管道，如何改变进程的标准输入以及如何改变进程的标准输出。

在这里可以将这所有的技巧结合在一起，编写一个通用的程序 pipe。它使用两个程序的名字作参数，例如：

```
pipe who sort
pipe ls head
```

程序的内在逻辑如下所示：

```
                        pipe(p)
                        fork()
                          |
  +-----------------------+-----------------------+
 child                                          parent
  |                                               |
 close(p[0])                                    close(p[1])
 dup2(p[1],1)                                   dup2(p[0],0)
 close(p[1])                                    close(p[0])
 exec "who"                                     exec "sort"
```

程序代码如下：

```
/* pipe.c
 * Demonstrates how to create a pipeline from one process to another
 * * Takes two args, each a command, and connects
 *   av[1]'s output to input of av[2]
 * * usage: pipe command1 command2
 *   effect: command1 | command2
 * * Limitations: commands do not take arguments
 * * uses execlp() since known number of args
 * * Note: exchange child and parent and watch fun
 */
#include <stdio.h>
#include <unistd.h>

  #define oops(m,x) { perror(m); exit(x); }

  main(int ac, char **av)
{
    int     thepipe[2],          /* two file descriptors   */
            newfd,               /* useful for pipes   */
            pid;                 /* and the pid   */

    if ( ac != 3 ){
      fprintf(stderr, "usage: pipe cmd1 cmd2\n");
      exit(1);
```

```
    }
    if ( pipe( thepipe ) == -1 )       /* get a pipe       */
        oops("Cannot get a pipe", 1);

    /*-----------------------------------------------------------------*/
    /* now we have a pipe, now let's get two processes        */

    if ( (pid = fork()) == -1 )        /* get a proc       */
        oops("Cannot fork", 2);

    /*-----------------------------------------------------------------*/
    /* Right Here, there are two processes        */
    /*              parent will read from pipe       */

    if ( pid > 0 ){             /* parent will exec av[2]       */
        close(thepipe[1]);    /* parent doesn't write to pipe       */

        if ( dup2(thepipe[0], 0) == -1 )
            oops("could not redirect stdin",3);

        close(thepipe[0]);    /* stdin is duped, close pipe       */
        execlp( av[2], av[2], NULL);
        oops(av[2], 4);
    }

    /* child execs av[1] and writes into pipe       */

    close(thepipe[0]);          /* child doesn't read from pipe       */

    if ( dup2(thepipe[1], 1) == -1 )
        oops("could not redirect stdout", 4);

    close(thepipe[1]);          /* stdout is duped, close pipe       */
    execlp( av[1], av[1], NULL);
    oops(av[1], 5);
}
```

程序 pipe.c 用了和 shell 一样的思路和技术来创建管道。但是 shell 并不像 pipe.c 一样运行外部程序。shell 首先创建管道，然后调用 fork 创建两个新进程，再将标准输入和输出重定向到创建的管道，最后再通过 exec 来执行两个程序。

10.6.4 技术细节：管道并非文件

管道在许多方面都类似于普通文件。进程使用 write 将数据写入管道，又通过 read 把数据读出来。像文件一样，管道是不带有任何结构的字节序列。另一方面，管道又与文件不同，例如文件的结尾是否也适用于管道呢？下列技术细节清楚地阐述了文件与管道的相同点与不同点。

1. 从管道中读数据

(1) 管道读取阻塞

当进程试图从管道中读数据时，进程被挂起直到数据被写进管道。那么如何避免进程永无止境地等待下去呢？

(2) 管道的读取结束标志

当所有的写者关闭了管道的写数据端时，试图从管道读取数据的调用返回 0，这意味着文件的结束。

(3) 多个读者可能会引起麻烦

管道是一个队列。当进程从管道中读取数据之后，数据已经不存在了。如果两个进程都试图对同一个管道进行读操作，在一个进程读取一些之后，另一个进程读到的将是后面的内容。它们读到的数据必然是不完整的，除非两个进程使用某种方法来协调它们对管道的访问。

2. 向管道中写数据

(1) 写入数据阻塞直到管道有空间去容纳新的数据

管道容纳数据的能力要比磁盘文件差的多。当进程试图对管道进行写操作的时候，此调用将挂起进程直到管道中有足够的空间去容纳新的数据。如果进程想写入 1000 个字节，而管道中现在只能容纳 500 个字节，那么这个写入调用就只好等待直到管道中再有 500 个字节空出来。如果某进程试图写 100 万字节，那结果会怎样？调用会不会永远等待下去呢？

(2) 写入必须保证一个最小的块大小

POSIX 标准规定内核不会拆分小于 512 字节的块。而 Linux 则保证管道中可以存在 4096 字节的连续缓存。如果两个进程向管道写数据，并且每一个进程都限制其消息不大于 512 字节，那么这些消息都不会被内核拆分。

(3) 若无读者在读取数据，则写操作执行失败

如果所有的读者都已将管道的读取端关闭，那么对管道的写入调用将会执行失败。如果在这种情况下，数据还可以被接收的话，它们会到哪里去呢？为了避免数据丢失，内核采用了两种方法来通知进程："此时的写操作是无意义的"。首先，内核发送 SIGPIPE 消息给进程。若进程被终止，则无任何事情发生。否则 write 调用返回 -1，并且将 errno 置为 EPIPE。

小　结

1. 主要内容

- 输入/输出重定向允许完成特定功能的程序通过交换数据来进行相互协作。
- Unix 默认规定程序从文件描述符 0 读取数据，写数据到文件描述符 1，将错误信息输出到文件描述符 2。这三个文件描述符称为标准输入、输出及标准错误输出。
- 当登录到 Unix 系统中，登录程序设置文件描述符 0、1、2。所有的连接、文件描述符都会从父进程传递给子进程。它们也会在调用 exec 时被传递。
- 创建文件描述符的系统调用总是使用最低可用文件描述符号。

- 重定向标准输入、输出以及错误输出意味着改变文件描述符 0、1、2 的连接。有很多种技术来重定向标准 I/O。
- 管道是内核中的一个数据队列，其每一端连接一个文件描述符。程序通过使用 pipe 系统调用创建管道。
- 当父进程调用 fork 的时候，管道的两端都被复制到子进程中。
- 只有有共同父进程的进程之间才可以用管道连接。

2. 下一步做什么

传统的 Unix 管道在进程间进行数据的单向传输。若两个进程希望来回传递数据又该如何？两个没有关联的进程或两个进程在不同的机器上，那该如何实现呢？在后面几章中，将更加细致地研究一下管道以及网络编程，使用管道的思路可以轻易地扩展到套接字(socket)上去。

3. 习题

10.1　符号>>可以告诉 shell 将输出添加到文件末尾上。shell 是使用自动添加模式(见第 5 章)，还是简单地寻找文件的结尾并从这里开始写数据？通过 shell 脚本设计一个试验来回答这个问题。

10.2　在 pipe.c 中，父进程运行着消费数据的程序，子进程运行着产生数据的程序。如果这两个程序的角色换一换，那么结果有何差异？将测试语句 if(pid>0)换为 if(pid == 0)，角色就可以被换过来。这将会导致什么情况的发生？为什么？

10.3　若需要 shell 支持管道，需要怎样做改动？首先，如何修改控制流来识别并处理以管道符号结尾的命令？其次，若有许多命令，它们之间通过管道符号来分割，那又该做怎样的修改？

10.4　在 pipe.c 中，读进程 sort 关闭了从父进程那里复制来的管道写数据端。修改代码让读进程不要关闭管道的写数据端。运行程序，并对结果做出解释。

10.5　在这一章的开始学习了将标准输入、输出重定向到文件的符号。知道重定向符号和文件名可以出现在命令行中的任意地方，并且重定向符号和文件名没有作为参数传递给程序。

前面编写的 shell 中，在控制流的什么地方识别出重定向的请求？何处实现这个重定向？若用户输入 set>varlist，结果又如何？这个 shell 会允许你将内部命令的输出重定向吗？如何将此功能加入自己编写的 shell 中呢？

10.6　若用户输入 sort<data>data，结果会如何呢？这个命令的问题何在？标准 Unix shell 如何处理这一问题呢？自己所写的 shell 又该如何去处理这个问题呢？

10.7　已经学习了如何将程序的标准输入输出连接到一个文件。上面所有的例子中都假设这里的文件是常规磁盘文件。那么是否可以将输入输出重定向到设备文件中呢？如果调用了 close(0)和 open("/dev/tty",0)，结果会如何呢？shell 是如何来处理命令 who>/dev/tty 的？

10.8 在 pipe.c 中，调用了 fork 和 exec 函数，但是并没有调用 wait。这是为什么？

10.9 讨论一下 dup 与 link 的共同点与区别。

4. 编程练习

10.10 修改脚本 watch.sh，使它可以有更多的功能。

(1) 除了打印所有用户的登录和注销情况外，新的版本要求可以通过命令行参数来传递一个包含要监控的用户列表文件。

(2) 修改后的程序仅仅当有新的用户登录或注销时，才将结果打印出来，而不是每次循环都打印一些内容。

(3) who 命令除了列出用户名之外还有登录时间、终端名称等。可能对某用户在哪个终端窗口登录并不感兴趣。修改程序使其仅仅当用户从登录状态变到非登录状态时才报告结果，而不去考虑终端的变化。

(4) 早期的 Unix 版本将数据存储在当前目录下的文件 prev 和 curr 中，并且在程序停止运行时，并没有对这两个文件做处理。这样的设计有哪些缺点？修改脚本，使其使用临时的文件，并在结束运行的时候删除它们。看一下如何使用 shell 中的 trap 命令以及 mktemp 命令。

10.11 修改 whotofile.c 程序，使其将 who 命令的输出添加到一个文件的末尾。确保在此文件不存在时，程序照样可以正常运行。

10.12 编写一个名为 sortfromfile.c 的程序，将 sort 命令的输入重定向，使其从文件中读入数据。文件名由命令行参数来指定。

10.13 扩展 pipe.c 程序来处理三节式管道。新版本的 Unix 程序可以接收三个程序名称作为参数，并让它们以管道的方式交互数据。命令

```
pipe3 who sort head
```

应该和 who|sort|head 起到相同的作用。

10.14 扩展上题的程序 pipe3 使其可以处理任意多的参数。

10.15 tee 工具允许重定向数据到文件并且将数据传递给另一个程序。例如：

```
who|tee userlist |sort > list2
```

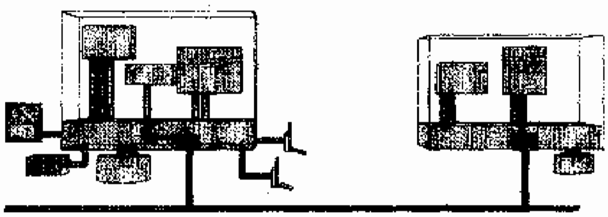
产生一个未排序文件和一个排序文件：userlist 和 list2。传给 tee 的参数是一个文件名，可以从参考手册得到更详细的信息。编写一个名叫 progtee 的程序来重定向数据到程序，同时通过管道将数据传递给另一个程序。例如命令：

```
who|progtee mail smith sort|progtee mail - s "hello"
    root > list2
```

将一个未排序的列表作为邮件发给 smith，将排好序的文件发给 root，且将一份排好序文件的备份放到文件 list2 中。

10.16　写数据到标准输出的程序往往并不在意文件描述符是联结到终端还是磁盘文件。本章中似乎暗示进程并没有方法来了解文件描述符的指向。其实这并不正确。库函数 isatty(fd)可以用来做判断：若文件描述符 fd 指向终端，则函数返回为 true，isatty 使用系统调用 fstat。阅读一下关于 fstat 的介绍，并且用它写一个函数 isaregfile(fd)，使其在 fd 指向常规磁盘文件时返回 true。

第 11 章　连接到近端或远端的进程：服务器与 Socket（套接字）



概念和技巧

- 客户/服务器模型
- 用管道来双向通信
- 协同进程(coroutines)
- 文件/进程的相似性
- 什么是 socket，为什么需要 socket，如何使用 socket
- 网络服务
- 用 socket 编写客户/服务器程序

相关系统调用和函数

- fdopen
- popen
- socket
- bind
- listen
- accept
- connect

11.1　产品和服务

像制造商利用传送带在工人之间传递产品一样，Unix 程序员可以通过管道来传送数字信息。

并非所有的商务过程都是像工厂的生产线一样是单向流动的，某些形式的通信方式是双向的。考虑一下衣服干洗店、律师还有兽医。你在把衣服交给干洗店，把宠物送到兽医那边，或是将文档通过 e-mail 发给律师之后，只需等待他们把处理好的东西发回，而并不需要像汽车工厂里的工人那样把车传递给下一个工人。在这个例子中，需要由其他人完

成的工作就称之为服务，而自己则是服务的客户。

上面的例子跟 Unix 有什么关系呢？Unix 中的管道可以把数据从一个进程传送到另外一个进程。进程和管道不但类似于一条生产线来完成产品，还类似于一个大的服务产业。本章将关注于进程间的数据通信，这也是客户/服务器编程的基础知识。

11.2 一个简单的比喻：饮料机接口

程序处理信息就像人们消耗饮料一样。以图 11.1 中的自动碳酸饮料机为例，在你投入硬币，按下按钮之后，饮料就自动流出。在此过程中，饮料机内的分配器在做什么工作呢？在饮料机内，应该有一桶碳酸水和另一捅浓缩的饮料汁水，当按下按钮时，将激活一个配制原材料并不断地传送出产生的碳酸饮料的过程。另一种方法则是只需要将一瓶预先配制好的碳酸饮料装到一个抽水泵上，按下按钮这个动作就简单地将饮料抽出并送给外面的杯子。

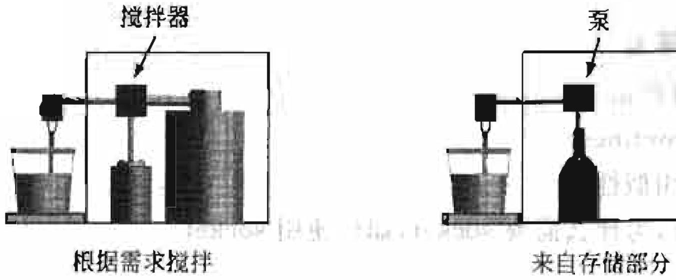


图 11.1　动态产生或来自静态饮料

就像碳酸饮料分配器一样，Unix 提供一个接口来处理可能来自不同数据源的数据，如图 11.2 所示。

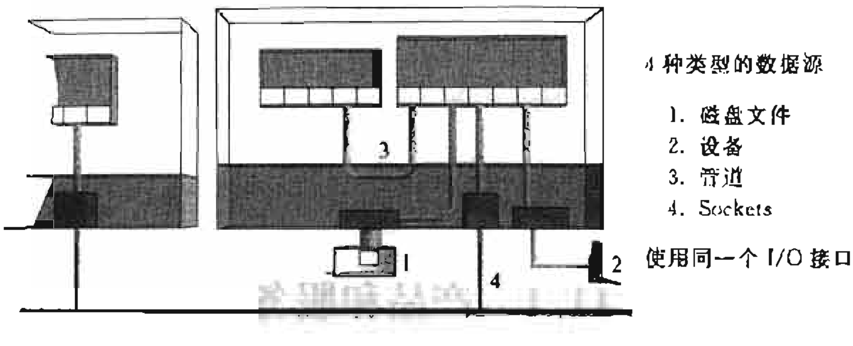


图 11.2　一个接口和不同的数据源

- (1,2)磁盘/设备文件

用 open 命令连接，用 read 和 write 传递数据。

- (3)管道

用 pipe 命令创建，用 fork 共享，用 read 和 write 传递数据。

• (4)Sockets

用 socket、listen 和 connect 连接，用 read 和 write 传递数据。

11.3　bc：Unix 中使用的计算器

儿乎每个版本的 Unix 都包含 bc 计算器，尽管这些计算器的版本有些差异。bc 计算器中包含变量、循环和函数的功能，并如在第 1 章中所看到的那样支持对长整数的处理。

```
$ bc
17^123
22142024630120207359320573764236957523345603216987331732240497016\
694729282299663749675090635587202539117092799463206393818799003\
722068558053628657356971 3
```

每行末尾的反斜线表示数字行的继续。

1. *bc 并不是一个计算器*

一个计算器程序分析它的输入，执行操作，然后将输出打印出来。大部分版本的 bc 程序都只分析输入，并不执行操作[①]。其实，bc 在内部启动了 dc 计算器程序，并通过管道与其进行通信。dc 是一个基于栈的计算器，它需要用户在指定具体的操作符之前，先输入所要操作的数据。例如，用户输入 2 2 + 来代表 2 加 2 的操作。

图 11.3 显示了 bc 如何来处理 2+2 的过程。用户输入 2+2，然后按回车。bc 从标准输入读取该表达式，分析出数据和操作符，接下来把一系列的命令"2"、"2"、"+"和 p 传给 dc，dc 则将数据入栈，运行加操作，最后把栈顶的数值送到标准输出。

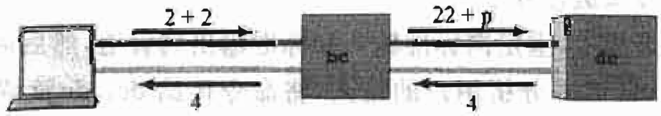


图 11.3　bc 和 dc 作为协同进程

bc 从连接到 dc 标准输出的管道上读取结果，再把结果转发给用户。这样的话，bc 甚至都不需要持有变量。如果用户输入 x=2+2，bc 告诉 dc 执行该操作并且把结果存到寄存器 x 中。命令 bc -c 可以显示分析器传给计算器的数据。就连 GNU 版本的 bc 也是把用户的输入转换成基于栈的后缀表达式。

2. *从 bc 方法中得到的思想*

(1) *客户/服务器模型*

bc/dc 程序对是客户/服务器模型程序设计的一个实例。dc 提供服务：计算。dc 所识别的语言是众所周知的逆波兰表示法。bc 和 dc 之间通过标准输入 stdin 和标准输出 stdout 进行通信。bc 提供用户界面，并使用 dc 提供的服务。这里 bc 被称为 dc 的客户。

---

① GNU 版本的 bc 是执行计算操作的。

这两个部分是根本上独立的程序。可以使用不同版本的 dc，这并不影响 bc 正常工作。类似地，可以编写一个图形界面的 bc，而仍用 dc 作为计算引擎。甚至可以用这样一个程序来替换 dc，该程序先分析 dc 所识别的语言，然后把它所要做的工作传给可能位于另一台更高速计算机上的程序。

(2) 双向通信

客户/服务器模型不同于生产线的数据处理模型，它要求一个进程既跟另一个进程的标准输入也要和它的标准输出进行通信。传统的 Unix 的管道只是单方向地传送数据①，图 11.3 给出了 bc 和 dc 之间的两个管道，其中上面的管道把一些计算命令传给 dc 的标准输入，下面的管道把 dc 的标准输出传给 bc。

(3) 永久性服务

bc 只是让单一的 dc 进程处于运行状态，这就不同于 shell 程序，这种程序中的每个用户命令都创建一个新的进程。bc 程序持续不断地和 dc 的同一个实例进行通信，把用户的输入转换成命令传给 dc。他们之间的关系并不同于标准函数中所使用的调用返回机制。

bc/dc 对被称之为协同进程(coroutines)以用来区别于子程序(subroutines)。两个程序都持续运行，当其中的一个程序完成自己的工作后将把控制权传给另一个程序。bc 的任务是分析输入及打印，而 dc 则负责计算。

11.3.1　编写 bc：pipe、fork、dup、exec

图 11.4 显示了内核将用户连接到 bc 并将 bc 连接到 dc 的数据连接。这里以该图作为编写下面代码的指南。

(1) 创建两个管道。

(2) 创建一个进程来运行 dc。

(3) 在新创建的进程中，重定向标准输入和标准输出到管道，然后运行 exec dc。

(4) 在父进程中，读取并分析用户的输入，将命令传给 dc，dc 读取响应，并把响应传给用户。

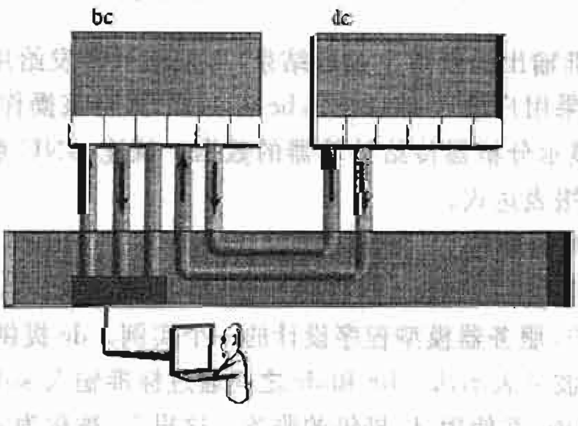


图 11.4　bc、dc 和内核

① 有一些管道也能双向传输数据(见小结中编程练习 11.11)。

下面是 tinybc. c 的源程序，这是一个简单版本的 bc，用 sscanf 分析输入，并通过两个管道与 dc 通信。

```
/**     tinybc.c        * a tiny calculator that uses dc to do its work
 **                     * demonstrates bidirectional pipes
 **                     * input looks like number op number which
 **                       tinybc converts into number \n number \n op \n p
 **                       and passes result back to stdout
 **
 **             +-----------+                +-----------+
 **     stdin  >0           >== pipetodc ====>           |
 **             |   tinybc  |                |    dc -   |
 **     stdout <1           <== pipefromdc ==<           |
 **             +-----------+                +-----------+
 **
 **                     * program outline
 **                         a. get two pipes
 **                         b. fork (get another process)
 **                         c. in the dc-to-be process,
 **                            connect stdin and out to pipes
 **                            then execl dc
 **                         d. in the tinybc-process, no plumbing to do
 **                            just talk to human via normal i/o
 **                            and send stuff via pipe
 **                         e. then close pipe and dc dies
 **                     * note: does not handle multiline answers
 **/
#include    <stdio.h>

#define oops(m,x)         { perror(m); exit(x); }

main()
{
    int     pid, todc[2], fromdc[2];            /* equipment */

    /* make two pipes */

    if ( pipe(todc) == -1 || pipe(fromdc) == -1 )
            oops("pipe failed",1);

    /* get a process for user interface */

    if ( (pid = fork()) == -1 )
                oops("cannot fork", 2);
```

```
    if ( pid == 0 )                          /* child is dc */
            be_dc(todc, fromdc);
    else {
            be_bc(todc, fromdc);             /* parent is ui */
            wait(NULL);                      /* wait for child */
    }
}

be_dc(int in[2], int out[2])
/*
 *      set up stdin and stdout, then execl dc
 */
{
    /* setup stdin from pipein */
        if ( dup2(in[0],0) == -1 )           /* copy read end to 0 */
                oops("dc: cannot redirect stdin",3);
        close(in[0]);                        /* moved to fd 0 */
        close(in[1]);                        /* won't write here */

    /* setup stdout to pipeout */
        if ( dup2(out[1], 1) == -1 )         /* dupe write end to 1 */
                oops("dc: cannot redirect stdout",4);
        close(out[1]);                       /* moved to fd 1 */
        close(out[0]);                       /* won't read from here */

    /* now execl dc with the - option */
        execlp("dc", "dc", "-", NULL );
        oops("Cannot run dc", 5);
}

be_bc(int todc[2], int fromdc[2])
/*
 *      read from stdin and convert into to RPN, send down pipe
 *      then read from other pipe and print to user
 *      Uses fdopen() to convert a file descriptor to a stream
 */
{
    int     num1, num2;
    char    operation[BUFSIZ], message[BUFSIZ], *fgets();
    FILE    *fpout, *fpin, *fdopen();

    /* setup */
    close(todc[0]);                          /* won't read from pipe to dc */
    close(fromdc[1]);                        /* won't write to pipe from dc */
```

```
    fpout = fdopen( todc[1],"w" );              /* convert file desc - */
    fpin = fdopen( fromdc[0], "r" );            /* riptors to streams */
    if ( fpout == NULL || fpin == NULL )
            fatal("Error convering pipes to streams");
    /* main loop */
    while ( printf("tinybc: "), fgets(message,BUFSIZ,stdin) !=
            NULL ){

            /* parse input */
            if ( sscanf(message,"%d%[- + */^]%d",
                    &num1,operation,&num2)!= 3){
                    printf("syntax error\n");
                    continue;
            }

            if ( fprintf( fpout , "%d\n%d\n%c\np\n", num1, num2,
                            * operation ) == EOF )
                                    fatal("Error writing");
            fflush( fpout );
            if ( fgets( message, BUFSIZ, fpin ) == NULL )
                    break;
            printf("%d %c %d = %s", num1, * operation , num2, message);
    }
    fclose(fpout);                               /* close pipe */
    fclose(fpin);                                /* dc will see EOF */
}

fatal( char * mess[] )
{
        fprintf(stderr, "Error: %s\n", mess);
        exit(1);
}
```

下面是 tinybc 的运行过程：

```
$ cc tinybc.c -o tinybc ; ./tinybc
tinybc: 2 + 2
2 + 2 = 4
tinybc: 55^5
55^5 = 503284375
tinybc:
```

仔细观察程序的输出，看一看哪一部分是来自于哪个程序。tinybc 产生了提示符并再一次显示出算术表达式。计算的结果是由 dc 产生的字符串，tinybc 只是从管道中读取该字符串然后把它传给输出。

11.3.2　对协同进程的讨论

其他的一些 Unix 工具是否也能被看成协同进程呢？sort 工具能否被当做协同进程使用？答案是否定的。在 sort 产生输出之前，它读取文件中所有的数据。通过管道来传送文件结尾标志的惟一途径是将写数据端关闭。一旦关闭了写的进程，就不能再传送其他需要排序的数据了。

从另一方面讲，dc 是逐行处理数据和命令的。与 dc 的交互很简单并可预测。当需要 dc 打印一个数据时，将会得到一行文本。当需要 dc 把数据压栈时，不会有响应被传回。

一个客户/服务器模型程序要成为协同系统必须有明确指明消息结束的方法，并且程序必须使用简单并可预测的请求和应答。

11.3.3　fdopen：让文件描述符像文件一样使用

在 tinybc.c 中使用了库函数 fdopen。fdopen 与 fopen 类似，返回一个 FILE * 类型的值，不同的是此函数以文件描述符而非文件作为参数。

使用 fopen 的时候，将文件名作为参数传给它。fopen 可以打开设备文件也可以打开常规的磁盘文件。如只知道文件描述符而不清楚文件名的时候可以使用 fdopen 命令。例如在管道的例子中，把一个通向管道的连接转换成 FILE * 类型值之后，就可以使用标准缓存的 I/O 操作来对其进行操作了。注意 tinybc.c 是如何使用 fprintf 和 fgets 来通过管道和 dc 进行通信的。

使用 fdopen 使得对远端的进程的处理就如同处理常规文件一样。下一节将剖析 popen 函数，此函数通过封装 pipe、fork、dup 和 exec 等系统调用使得对程序和文件的操作变成了一回事。

11.4　popen：让进程看似文件

这一节将继续学习一个程序如何通过和另外一个进程通信来得到服务。本节还将剖析 popen 库函数，理解 popen 及其工作原理，最后给出 popen 实现版本。

11.4.1　popen 的功能

fopen 打开一个指向文件的带缓存的连接：

```
FILE * fp;                          /* a pointer to a struct */
fp = fopen("file1","r");            /* args are filename,connection type */
c = getc(fp);                       /* read char by char */
fgets(buf,len,fp);                  /* line by line */
fscanf(fp,"%d%d%s",&x,&y,x);        /* token by token */
fclose(fp);                         /* close when done */
```

fopen 需要两个字符串变量作为参数：文件名和连接类型(例如："r"、"w"、"a"、…)。popen 看上去跟 fopen 很类似。popen 打开一个指向进程的带缓冲的连接：

```
FILE * fp;                      /* same type of struct */
fp = popen("ls","r")            /* args are program name, connection type */
fgets(buf,len,fp);              /* exactly the same functions */
pclose(fp);                     /* close when done */
```

图 11.5 显示了 popen 和 fopen 之间的相似性。两者使用相同的语法格式，并具有相同的返回值类型。popen 的第一个参数是要打开的命令的名称；它可以是任意的 shell 命令。第二个参数可以是“r”或“w”，但决不会是“a”。

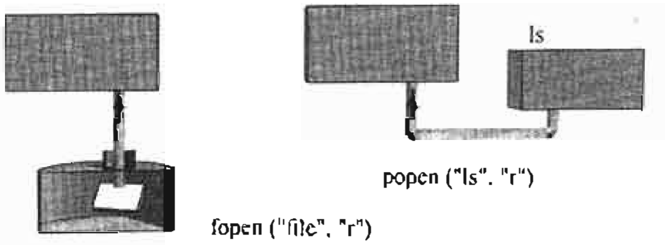


图 11.5 fopen 和 popen

下面的程序将 who|sort 作为数据源，通过 popen 来获得当前用户排序列表：

```
/* popendemo.c
 *      demonstrates how to open a program for standard i/o
 *      important points:
 *              1. popen() returns a FILE *, just like fopen()
 *              2. the FILE * it returns can be read/written
 *                 with all the standard functions
 *              3. you need to use pclose() when done
 */
#include    <stdio.h>
#include    <stdlib.h>

int main()
{
    FILE    * fp;
    char    buf[100];
    int     i = 0;

    fp = popen( "who|sort", "r" );          /* open the command */

    while ( fgets( buf, 100, fp ) != NULL )  /* read from command */
        printf("%3d %s", i++, buf );        /* print data */

    pclose( fp );                           /* IMPORTANT! */
    return 0;
}
```

第二个例子将 popen 和邮件程序相连接，用来提示用户一些系统故障：

```
/* popen_ex3.c
 *      shows how to use popen to write to a process that
 *      reads from stdin. This program writes email to
 *      two users. Note how easy it is to use fprintf
 *      to format the data to send.
 */

#include  <stdio.h>

main()
{
     FILE *fp;

     fp = popen( "mail admin backup", "w" );
     fprintf( fp, "Error with backup!! \n" );
     pclose( fp );
}
```

pclose 命令是必须的。

当完成对 popen 所打开连接的读写后，必须使用 pclose 关闭连接，而不能用 fclose。进程在产生之后必须等待退出运行，否则它将成为僵尸进程(zombie)。而 pclose 中调用了 wait 函数来等待进程的结束。

11.4.2　实现 popen：使用 fdopen 命令

popen 是如何工作的？如何来实现它？popen 运行了一个程序并返回指向该程序标准输入或标准输出的连接。

这里需要一个新的进程来运行程序，所以要用到 fork 命令。需要一个指向该进程的连接，因此需要使用管道。并且使用 fdopen 命令将一个文件描述符定向到缓冲流中。最后，在该进程中要能够运行任何 shell 命令，所以要用到 exec。但是会运行什么程序呢？惟一能够运行任意 shell 命令的程序是 shell 本身即/bin/sh。为了使程序员可以方便的使用，sh 支持 -c 选项，用以告诉 shell 执行某命令然后退出。例如：

```
sh -c "who|sort"
```

告诉 sh 执行命令行 who|sort，如图 11.6 所示。

这里合并了 pipe、fork、dup2 和 exec 等系统调用，其流程如下：

```
                        pipe(p)
                        fork()
                          |
      +-------------------+-------------------+
close(p[1]);                              close(p[0]);
```

```
fp = fdopen(p[0],"r");                          dup(p[1],1);
return fp;                                      close(p[1]);
                                          exec("/bin/sh","sh","-c",cmd,NULL);
```

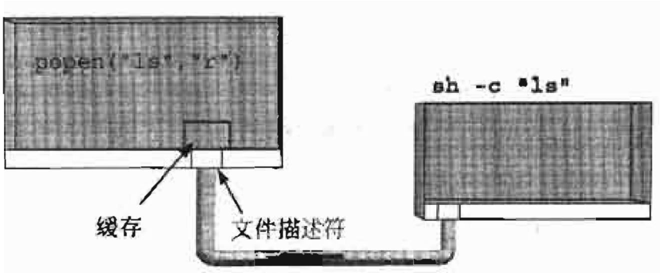


图 11.6　从 shell 命令中读取

下面的程序 popen.c 是上述流程的一个实现：

```
/* popen.c - a version of the Unix popen() library function
 *      FILE * popen( char * command, char * mode )
 *              command is a regular shell command
 *              mode is "r" or "w"
 *              returns a stream attached to the command, or NULL
 *              execls "sh" "-c" command
 *    todo: what about signal handling for child process?
 */
#include    <stdio.h>
#include    <signal.h>

#define READ     0
#define WRITE    1

FILE * popen(const char * command, const char * mode)
{
    int       pfp[2], pid;                    /* the pipe and the process */
    FILE      * fdopen(), * fp;               /* fdopen makes a fd a stream */
    int       parent_end, child_end;          /* of pipe */

    if ( * mode == 'r' ){                     /* figure out direction */
            parent_end = READ;
            child_end = WRITE ;
    } else if ( * mode == 'w' ){
            parent_end = WRITE;
            child_end = READ ;
    } else return NULL ;

    if ( pipe(pfp) == -1 )                    /* get a pipe */
            return NULL;
```

```
    if ( (pid = fork()) == -1 ){                    /* and a process */
            close(pfp[0]);                          /* or dispose of pipe */
            close(pfp[1]);
            return NULL;
    }

    /* ---------------- parent code here ------------------- */
    /*  need to close one end and fdopen other end */

    if ( pid > 0 ){
            if (close( pfp[child_end] ) == -1 )
                    return NULL;
            return fdopen( pfp[parent_end] , mode);/* same mode */
    }

    /* ---------------- child code here ------------------- */
    /*  need to redirect stdin or stdout then exec the cmd */

    if ( close(pfp[parent_end]) == -1 )/* close the other end */
            exit(1);                                /* do NOT return */

    if ( dup2(pfp[child_end], child_end) == -1 )
            exit(1);

    if ( close(pfp[child_end]) == -1 )              /* done with this one */
            exit(1);
                                                    /* all set to run cmd */
    execl( "/bin/sh", "sh", "-c", command, NULL );
    exit(1);
}
```

该版本的 popen 对信号不做任何处理。这是不是有问题?

11.4.3 访问数据:文件、应用程序接口(API)和服务器

fopen 从文件获得数据,而 popen 从进程获得数据。这里主要关注一下数据访问的普遍问题并对三种实现方法进行比较。将以获取登录系统的用户列表为例来比较三种访问数据的方法。

• 方法 1:从文件获取数据

可以通过读取文件来获取数据。第 2 章所写的 who 程序就是从 utmp 文件中读取数据的。基于文件的信息服务并不是很完美。客户端程序依赖于特定的文件格式和结构体中的特定成员名称。下面的 Linux 头文件定义中 utmp 结构体的几行代码清楚地展示了这一点。

```
/*Backwards compatibility hacks. */
#define ut_name ut_user
```

• 方法 2：从函数获取数据

可以通过调用函数来得到数据。一个库函数用标准的函数接口来封装数据的格式和位置。Unix 提供了读取 utmp 文件的函数接口。getutent 的帮助信息描述了读取 utmp 数据库函数的细节。这样的话，就算底层的存储结构变化了，使用这个接口的程序仍能正常工作。

使用基于应用程序接口(API)的信息服务也并不一定是最好的方法。有两种方法可以使用库函数。一个程序可以使用静态连接来包含实际的函数代码。但是这些函数有可能包含的并不是正确的文件名或文件格式。另一方面，一个程序可以调用共享库中的函数，但是这些共享库也并不是安装在所有的系统上，或者其版本并不是程序所要使用的版本。

• 方法 3：从进程获取数据

第三种方法是从进程中读取数据。bc/dc 和 popen 例子显示了如何创建一个进程到另外一个进程的连接。一个要得到用户列表的程序可以使用 popen 来建立与 who 程序的连接。由 who 命令来负责使用正确的文件名和文件格式以及正确的库函数，而不是你的程序。

调用独立的程序获得数据还有其他的好处。服务器程序可以使用任何程序设计语言编写：shell 脚本、C、Java 或是 Perl 都可以。以独立程序的方式实现系统服务的最大好处是客户端程序和服务器端程序可以运行在不同的机器上。所有要做的只是和不同机器上的一个进程相连接。

11.5　socket：与远端进程相连

管道使得进程向其他进程发送数据就像向文件发送数据一样容易，但是管道具有两个重大的缺陷。管道在一个进程中被创建，通过 fork 来实现共享。因此，管道只能连接相关的进程，也只能连接同一台主机上的进程。Unix 提供了另外一种进程间的通信机制——socket。

socket 允许在不相关的进程间创建类似管道的连接，甚至可以通过 socket 连接其他主机上的进程(如图 11.7 所示)。本节将学习 socket 的基础知识，理解如何用 socket 连接不同主机上的客户端和服务器端。其思想就跟打电话查询当地时间一样简单。

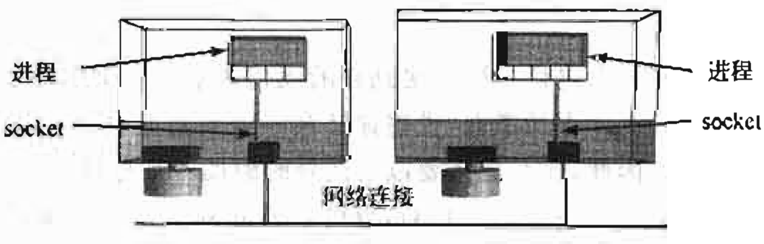


图 11.7　连接到远端的进程

11.5.1　类比："电话中传来声音：现在时间是……"

许多城市都设有提供时间服务的电话号码。只要拨打该号码，机器将负责告诉你该城市的时间。它是如何工作的呢？如果想建立自己的时间服务，该如何做呢？可以采用图 11.8 中所描述的比较简单的方法。你坐在办公室里，将一个钟挂在墙上来扮演提供时间服务的服务器。你所要遵循的步骤和 Unix 中基于 socket 建立时间服务器的步骤完全一致。下面，将详细讨论这些步骤。

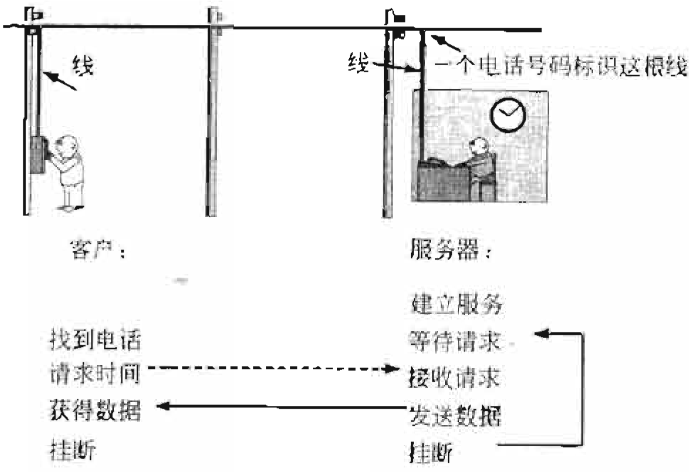


图 11.8　时间服务

1. *建立服务及与服务相关的操作*

一旦买好并安装了你自己的时钟，如何建立和操作时间服务器呢？

(1) 建立服务

建立服务包含以下 3 个步骤。

- 获取一根电话线

首先，需要一根接自公用电话网上的电话线来连接办公桌旁墙上的插座。该电话线和插座使你可以连接到电话网上，接下来外面的呼叫可以被传送到你的办公桌。这里的插座通常被称为通信端点(endpoint of communication)。下一次如果需要在家里安装电话线，可以向电话局申请安装一个通信端点。

- 为电话线申请号码

客户端需要通过呼叫一个电话号码连接到你的通信端点。电话网络以电话号码来区分每个墙上的插座。从这个类比本身考虑，设想你是在一个大的商务系统中，除了时间服务以外还需提供其他的服务。因此，每个插座要以电话号码附加一个分机号码来标识。

例如：你的号码可能是 617—999—1234，分机号码是 8080，电话号码标识了你的办公室所在的大楼的号码，扩展号码 8080 标识了该楼中每个具体的电话。一个号码来标识大楼，一个分机号码来标识你的服务，这是一个重要的机制。

- 处理接入电话

你可能使用的是无来电显示的电话。你的服务不需要上述类型的电话服务。但必须通

知电话网络让你的电话线可以接收接入呼叫。你可以为接入呼叫设置一个队列，并用一条消息来提示拨打者对你来说他们呼叫的重要程度，然后播放一段音乐。在 socket 中也使用了队列的思想，当然不会播放音乐。

(2) 与服务相关的操作

与时间服务相关的操作是包含以下 3 个步骤的一个循环。

- 等待呼叫

等待在那儿，不做任何事情直到有呼叫进来。从技术的角度讲，即被阻塞在一个呼叫上。当一个呼叫进来，你解除阻塞并接收呼叫。

- 提供服务

在这个例子中，你看一下钟，然后通过电话线把时间告诉对方。

- 挂断

已经完成了对该呼叫的工作，挂断电话。

上述 6 个步骤中，3 个用来建立服务，3 个用来对应每个接入呼叫。这 6 个步骤就是对在电话网上运行时间服务的详细描述。

2. 使用服务

客户端如何使用所提供的服务呢？每个客户端都遵循以下 4 个步骤。

(1) 获取一根电话线

客户端同样需要一个通信端点。客户端还要从电话网络申请一个电话号码。

(2) 连接到具体号码

客户端使用这根电话线向电话网络请求建立一个到特定线路的连接。客户端拨打大楼的电话及你办公室的分机，并与其相连。其中的商务楼的号码和分机号码的组合被称为服务的网络地址。从技术的角度讲，商务楼号码是主机地址，分机号码是端口号或者称为端口。在前面的例子中，主机号是 617—999—1234，端口是 8080。

(3) 使用服务

两个通信端点(客户端的和服务器端的)现在已经建立了连接。任何一方可以通过该连接向另外的通信端点发送数据。以时间服务器为例，服务器通过线路发送数据，而客户端则接收信息。像目录编排这样更加复杂的服务就需要服务器和客户端之间更复杂的交互。本书在后面将分析更复杂的服务。

(4) 挂断电话

交互已经完成。客户端可以挂断电话。

3. 重要概念

时间服务器例子包含了 socket 编程中所要涉及的 4 个重要的概念。

(1) 客户和服务器

已经不止一次地讨论该问题了。服务器是提供服务的程序。在 Unix 中，服务器是一个程序不是一台计算机。服务器进程等待请求，处理请求，然后循环回去等待下一个请求。客户端进程则不需要循环，它只需建立一个连接，与服务器交换数据，然后继续自己的工作。

(2) 主机名和端口

运行于因特网上的服务器其实是某台计算机上运行的一个进程。这里计算机被称为主

机。机器通常被指定一个名字如 sales. xyzcorp. com，这被称为该主机的名字。服务器在该主机上拥有一个端口。主机和端口的组合才标识了一个服务器。

(3) 地址族

你的时间服务必须拥有一个电话号码，它还可能有街道地址和邮编，甚至可能有经度和纬度或是其他集合的数据等属性。上述每个集合的数据都是你的服务的地址。但是如用经度和纬度来代替电话号码中的电话号码和扩展号码的话，它们有可能不能正常运作。

上面的每个地址分别属于不同的地址族。电话号码和分机号码是电话网络地址族的地址，这里可以用 AF_PHONE 来标识。类似地，经度和纬度在全球坐标系统地址族中才有意义，并可以用 AF_GLOBAL 来标识。

(4) 协议

协议是服务器和客户之间交互的规则。在时间服务器的例子中，协议很简单：客户呼叫，服务器回答，给出时间信息后挂断。

如果运行的是一个查号辅助服务，又将如何呢？协议将会复杂一点。服务器需要回答和发送初始欢迎信息(例如："欢迎访问查号辅助系统，请问您在哪个城市?")。客户端给出城市的名字后，服务器将询问所查名字(例如："您需要查询什么?")。客户端以某公司或个人的名字作应答。这时服务器给出所要请求的电话号码或此城市不存在所查询的名字的消息。一些查号辅助服务器还提供付费的电话服务。消息的交互遵循查号辅助协议(directory-assistance protocol)，本章称其为 DAP。每个客户/服务器模型都必须定义这样一个协议。

11.5.2 因特网时间、DAP 和天气服务器

时间服务器和查号辅助服务器的例子不仅仅是用于教学的比喻，它们在因特网中存在着广泛的应用。试一下下面的命令：

```
$ telnet mit.edu 13
Trying 18.7.21.69...
Connected to mit.edu
Escape character is '^]'.
Mon Aug 13 22:36:44 2001
Conncetion closed by foreign host.
$
```

位于 MIT 的某台主机上有一台时间服务器，它在 13 号端口等待请求。当用 telnet 和该服务器连接的时候，服务器接收请求，检查系统时钟，通过网络送回当前的时间，然后挂断连接。跟前面的时间服务的例子完全相同，它们甚至使用了相同的协议。试一下连接到其他主机的 13 号端口。可以得到世界上其他任何机器上的时间。

telnet 程序就像一部电话。它和远端主机的某个端口建立连接，然后把你键盘上的输入数据通过连接发送出去，并将通过连接接收到的数据显示在屏幕上。

那查号辅助服务又如何呢？查号辅助服务器通常在 79 号端口监听。例如，交互过程如

下所示：

```
$ telnet princeton.edu 79
Trying 128.112.128.81...
Connected to princeton.edu.
Escape character is '^]'.
Smith
------------------------------------------------------------------------
       alias: 000012345
        name: Waldo Smith
  department: Special Student
       email: waldos@Princeton.EDU
    emailbox: waldos@mail.Princeton.EDU
       netid: waldos
------------------------------------------------------------------------
       alias: 000333333
        name: Ignatz E Smith
  department: Undergraduate Class of 1997
       email: ismith@Princeton.EDU
    emailbox: ismith@mail.Princeton.EDU
       netid: ismith
           ⋮
```

当一个请求到来时，服务器接收该请求。协议中规定了客户必须输入用户名并以回车结束。服务器将所有的匹配入口传回，然后挂断连接。

天气服务又是怎样呢？试一试下面的命令：

```
telnet rainmaker.wunderground.com 3000
```

该天气服务的协议更加复杂，不过界面却友好得多。

11.5.3　服务列表：众所周知的端口

如何知道端口 13 是时间服务端口而 79 是目录辅助服务呢？同样的道理，在美国每个人都知道号码 911 是紧急服务，号码 411 是查号服务，这些就是众所周知的端口。在文件/etc/services 中定义了众所周知服务端口号的列表：

```
$ more /etc/services
#         $ NetBSD:services, v 1.18 1996/03/26 00:07:58 mrg Exp $
# Network services, Internet style
#
# Note that it is presently the policy of IANA to assign a single
# well-known port number for both TCP and UDP;hence, most entries
# here have two entries even if the protocol doesn't support UDP
# operations.
```

```
# Updateed from RPC 1340, "Assigned Numbers"(July 1992). Not all
# ports are included, only the more common ones.
#
#          from:@(#)services     5.8 (Berkeley) 5/9/91
#
tcpmux     1/tcp # TCP port service multiplexer
echo       7/tcp
echo       7/udp
discard    9/tcp      sink null
discard    9/udp      sink null
systat     11/tcp     users
datetime   13/tcp
datetime   13/udp
--more--(13%)
```

从列表中可以看出时间服务是端口 13。仔细研究该文件可以看到因特网主机上的标准服务，如 ftp、telnet、finger 和 http 的端口。

所有这些运行在因特网主机上的服务实际上都是基于前面给出的时间服务器模型的思想和步骤的。这里将把这些思想应用于 Unix 的系统调用，从而编写自己的时间服务器以及客户端版本。

11.5.4 编写 timeserv.c：时间服务器

上面提到的基于电话的时间服务涉及 6 个步骤。每个步骤与一个系统调用相对应。对应关系如下所示：

行　　为	系 统 调 用
1. 获取电话线	socket
2. 分配号码	bind
3. 允许接入调用	listen
4. 等待电话	accept
5. 传送数据	read/write
6. 挂断电话	close

程序如下：

```
/* timeserv.c a socket-based time of day server
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include  <netinet/in.h>
#include  <netdb.h>
#include  <time.h>
#include  <strings.h>

#define   PORTNUM  13000  /* our time service phone number */
#define   HOSTLEN  256
#define   oops(msg)       { perror(msg) ; exit(1) ; }

int main(int ac, char * av[])
{
    struct  sockaddr_in    saddr;                /* build our address here  */
    struct  hostent       * hp;                  /* this is part of our */
    char    hostname[HOSTLEN];                   /* address */
    int     sock_id,sock_fd;                     /* line id, file desc */
    FILE    * sock_fp;                           /* use socket as stream */
    char    * ctime();                           /* convert secs to string */
    time_t  thetime;                             /* the time we report */

  /*
   * Step 1: ask kernel for a socket
   */

    sock_id = socket( PF_INET, SOCK_STREAM, 0 );   /* get a socket */
    if ( sock_id == -1 )
            oops( "socket" );

  /*
   * Step 2: bind address to socket. Address is host,port
   */

    bzero( (void *)&saddr, sizeof(saddr) );        /* clear out struct */

    gethostname( hostname, HOSTLEN );              /* where am I ? */
    hp = gethostbyname( hostname );                /* get info about host */
                                                   /* fill in host part */
    bcopy( (void *)hp->h_addr, (void *)&saddr.sin_addr,
            hp->h_length);
    saddr.sin_port = htons(PORTNUM);               /* fill in socket port */
    saddr.sin_family = AF_INET ;                   /* fill in addr family */

    if ( bind(sock_id, (struct sockaddr *)&saddr, sizeof(saddr)) != 0 )
            oops( "bind" );

  /*
   * Step 3: allow incoming calls with Qsize=1 on socket
```

```
  */

  if ( listen(sock_id, 1) != 0 )
          oops( "listen" );

 /*
  * main loop: accept(), write(), close()
  */

  while ( 1 ){
        sock_fd = accept(sock_id, NULL, NULL);       /* wait for call */
          printf("Wow! got a call! \n");
        if ( sock_fd == -1 )
                oops( "accept" );                    /* error getting calls */

        sock_fp = fdopen(sock_fd,"w");               /* we'll write to the */
        if ( sock_fp == NULL )                       /* socket as a stream */
                oops( "fdopen" );                    /* unless we can't */

        thetime = time(NULL);                        /* get time */
                                                     /* and convert to strng */

        fprintf( sock_fp, "The time here is .." );
        fprintf( sock_fp, "%s", ctime(&thetime) );
        fclose( sock_fp );                           /* release connection */
  }
}
```

下面将对程序如何工作给出解释。

- 步骤 1：向内核申请一个 socket

socket 是一个通信端点。就像位于墙上的电话插座一样，socket 是产生呼叫和接收呼叫的地方。系统调用 socket 创建一个 socket。

socket		
目标	建立一个 socket	
头文件	#include <sys/types.h> #include <sys/socket.h>	
函数原型	sockid=socket(int domain,int type,int protocol)	
参数	domain	通信域 PF_INET 用于 Internet socket
	type	socket 的类型。SOCK_STREAM 跟管道类似
	protocol	协议 socket 中所用的协议，默认为 0
返回值	−1	遇到错误
	sockid	成功返回

socket 调用创建一个通信端点并返回一个标识符。有很多种类型的通信系统，每个被称为一个通信域。Internet 本身就是一个域。在后面会看到 Unix 内核是另一个域。Linux 支持好几个其他域的通信。

socket 的类型指出了程序将要使用的数据流类型。SOCK_STRAEAM 类型跟双向的管道类似。数据作为连续的字节流从一端写入，再从另一端读出。后面的章节中将介绍 SOCK_DGRAM 类型。

函数中最后的参数 protocol 指的是内核中网络代码所使用的协议，并不是客户端和服务器之间的协议。一个为 0 的值代表选择标准的协议。

• 步骤 2：绑定地址到 socket 上，地址包括主机、端口

下一个步骤是把一个网络地址分配给 socket。在 Internet 域中，地址由主机和端口构成。这里不能使用端口 13，因为该端口已经为时间服务器保留。这里可使用端口 13000 来代替。可以为你的服务器端口选择任意的号码，只是该号码不要太小且不能已经被占用。低端口号可能已经被系统服务所占用，而不能再被普通用户使用。请检查系统中端口的限制范围。端口号是一个 16 位的数值，所以有很多端口号可用。系统调用 bind 如下所示。

	bind	
目标	绑定一个地址到 socket	
头文件	#include <sys/types.h> #include <sys/socket.h>	
函数原型	result=bind(int sockid, struct sockaddr * addrp, socklen_t addrlen)	
参数	sockid	socket 的 id
	addrp	指向包含地址结构的指针
	addrlen	地址长度
返回值	−1	遇到错误
	0	成功

bind 调用把一个地址分配给 socket。该地址的分配就类似于把一个电话号码分配给墙上的一个插座；当进程要与服务器连接的时候，它们就使用该地址。每个地址族都有自己的格式。因特网地址族(AF_INET)使用主机和端口来标志。地址就是一个以主机和端口为成员的结构体。自己写的程序应首先初始化该结构的成员，然后再填充具体的主机地址和端口号，最后填充地址族。关于建立上述数据的具体函数，可参阅帮助手册。

当所有的部分被填充了之后，地址已经被绑定到该 socket 上。其他类型的 socket 会使用包含不同成员的地址。

• 步骤 3：在 socket 上，允许接入呼叫并设置队列长度为 1

服务器接收接入的呼叫，所以这里的程序必须使用 listen。

listen		
目标	监听 socket 上的连接	
头文件	#include <sys/socket. h>	
函数原型	result=listen(int sockid,int qsize)	
参数	sockid	接收请求的 socket
	qsize	允许接入连接的数目
返回值	−1	遇到错误
	0	成功

listen 请求内核允许指定的 socket 接收接入呼叫。并不是所用类型的 socket 都能接收接入呼叫。但 SOCK_STREAM 类型是可以的。第二个参数指出接收队列的长度。在本章的程序中请求的是一个长度为 1 的队列。队列最大长度则取决于具体 socket 的实现。

• *步骤 4：等待/接收呼叫*

一旦 socket 被建立并被分配一个地址，而且准备等待接收呼叫，程序即将开始工作。服务器等待直到呼叫到来。它使用系统调用 accept 来接收调用。

accept		
目标	接收 socket 上的一个连接	
头文件	#include <sys/types. h> #include <sys/socket. h>	
函数原型	fd=accept(int sockid,struct sockaddr * callerid,socklen_t * addrlenp)	
参数	sockid	接收该 socket 上的呼叫
	callerid	指向呼叫者地址结构的指针
	addrlenp	指向呼叫者地址结构长度的指针
返回值	1	遇到错误
	fd	用于读写的文件描述符

accept 阻塞当前进程，一直到指定 socket 上的接入连接被建立起来，然后 accept 将返回文件描述符，并用该文件描述符来进行读写操作。此文件描述符实际上是连到呼叫进程的某个文件描述符的一个连接。

accept 支持一种类型的呼叫者的 ID。在呼叫发起者一边，socket 有自己的地址，例如对于因特网连接，地址就是主机地址和端口号。如果 callerid 和 addrlenp 指针不为空的话，内核将把呼叫者地址填充到 callerid 所指向的结构中，并把该结构的长度填充到 addrlenp 所指向的内存单元中。

就像人们使用来电显示的信息来决定如何处理打入的电话一样，一个网络程序可以使用呼叫进程的地址来决定如何处理该连接。

• *步骤 5：传输数据*

accept 调用所返回的文件描述符是一个普通文件的描述符。对它的操作从第 2 章中学习完 open 调用之后一直在使用。程序 timeserv. c 用 fdopen 将文件描述符定向到缓存的数

据流，以便于使用 fprintf 调用来进行输出。在以前只能使用 write 来完成这项工作。

- 步骤 6：关闭连接

accept 所返回的文件描述符可以由标准的系统调用 close 关闭。当一端的进程关闭了该端的 socket，若另一端的进程在试图读数据的话，它将得到文件结束标记。这跟管道的工作原理类似。

11.5.5　测试 timeserv.c

下面可以编译并运行时间服务器：

```
$ cc timeserv.c - o timeserv
$ timeserv&
29362
$
```

启动服务以"&"符号结尾，所以 shell 将运行它但不调用 wait 调用。服务将阻塞在 accept 调用上。这里可以用 telnet 连接到服务器上：

```
$ telnet 'hostname' 13000
Trying 123.123.123.123
Connected to somesite.net
Escape character is '^]'.
Wow! got a call!
The time here is ..Tue Aug 14 11:36:30 2001
Connection close by foreign host.
$
$ telnet 'hostname' 13000
Trying 123.123.123.123
Connected to somesite.net
Escape character is '^]'.
Wow! got a call!
The time here is ..Tue Aug 14 11:36:53 2001
Connection close by foreign host.
$
```

上面的程序建立了两个连接，并且服务器响应了每个连接。服务器将持续运行，直到使用 kill 命令来结束其运行。

```
$ kill 29362
```

对于此服务器而言，telnet 程序是其客户。但它并不总是适合连接到任何服务器上的。这里将针对该服务器编写一个特殊的客户端程序。

11.5.6　编写 timeclnt.c：时间服务客户端

基于电话的时间服务客户端的实现包含 4 个步骤，每个步骤对应于一个系统调用。

行　　为	系 统 调 用
1. 获取一根电话线	socket
2. 呼叫服务器	connect
3. 传送数据	read/write
4. 挂断电话	close

下面是这个程序：

```
/* timeclnt.c - a client for timeserv.c
 *              usage: timeclnt hostname portnumber
 */

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/socket.h>
#include   <netinet/in.h>
#include   <netdb.h>

#define   oops(msg)         { perror(msg); exit(1); }

main(int ac, char *av[])
{
    struct sockaddr_in servadd;         /* the number to call */
    struct hostent      *hp;            /* used to get number */
    int     sock_id, sock_fd;           /* the socket and fd */
    char    message[BUFSIZ];            /* to receive message */
    int     messlen;                    /* for message length */

  /*
   * Step 1: Get a socket
   */

    sock_id = socket( AF_INET, SOCK_STREAM, 0 );      /* get a line */
    if ( sock_id == -1 )
        oops( "socket" );                             /* or fail */

  /*
   * Step 2 : connect to server
   *         need to build address (host,port) of server first
   */

    bzero( &servadd, sizeof( servadd ) );    /* zero the address */

    hp = gethostbyname( av[1] );             /* lookup hosts ip # */
    if (hp == NULL)
        oops(av[1]);                         /* or die */

    bcopy(hp->h_addr, (struct sockaddr *)&servadd.sin_addr,
```

```
        hp->h_length);

    servadd.sin_port = htons(atoi(av[2])); /* fill in port number */

    servadd.sin_family = AF_INET ;          /* fill in socket type */
                                            /* now dial */
    if ( connect(sock_id,(struct sockaddr *)&servadd,
        sizeof(servadd)) != 0)
      oops( "connect" );

  /*
   * Step 3: transfer data from server, then hangup
   */

    messlen = read(sock_id, message, BUFSIZ);      /* read stuff */
    if ( messlen == - 1 )
        oops("read") ;
    if ( write( 1, message, messlen ) != messlen ) /* and write to */
        oops( "write" );                           /* stdout */
    close( sock_id );
}
```

下面是对该程序的解释。

- 步骤 1：向内核请求建立 socket

客户端需要一个 socket 跟网络相连，就像时间服务中的客户端需要一条电话线跟电话网络相连一样。客户端必须建立 Internet 域（AF_INET）socket，并且它还必须是流 socket（SOCK_STREAM）。

- 步骤 2：与服务器相连

客户端需要连接到时间服务器。connect 系统调用的作用实际上与打电话类似。

connect		
目标	连接到 socket	
头文件	#include <sys/types.h> #include <sys/socket.h>	
函数原型	result=connect(int sockid, struct sockaddr * serv_addrp, socklen_t addrlen);	
参数	sockid	用于建立连接的 socket
	serv_addrp	指向服务器地址结构的指针
	addrlen	结构的长度
返回值	−1	遇到错误
	00	成功

connect 调用试图把由 sockid 所标识的 socket 和由 serv_addrp 所指向的 socket 地址相连接。如果连接成功的话，connect 返回 0。而此时，sockid 是一个合法的文件描述符，可以用来进行读写操作。写入该文件描述符的数据被发送到连接的另一端的 socket，而从另一端

写入的数据将从该文件描述符读取。

- 步骤 3 和 4：传送数据和挂断

在成功连接之后，进程可以从该文件描述符读写数据，就像与普通的文件或管道相连接一样。在时间服务的客户/服务器例子中，timeclnt 只是从服务器读取一行数据。

读取时间之后，客户端关闭文件描述符然后退出。若客户端退出而不关闭描述符，内核将完成关闭文件描述符的任务。

11.5.7 测试 timeclnt.c

大家已经有好几页没有看到插图了，大概已经忘记这些代码的任务了吧。图 11.9 将会提醒我们。服务器进程运行在一台机器上，而客户端程序在另一台机器上通过网络与服务器连接。服务器通过 write 调用来发送数据，客户端通过 read 调用来接收消息。

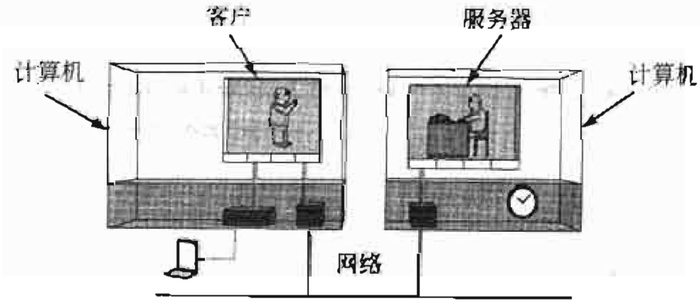


图 11.9　位于不同机器上的进程

对上面这段代码真实的测试需要在不同机器上运行这两个程序。我不太确定下面写在书上的测试情况是否足够明确，不过大家可以作为参考：

```
$ hostname                                  # 检查当前机器
computer1.mysite.net                        # 第一台机器
$ cc timeserv.c -o timeserv                 # 建立服务器
$ ./timeserv &                              # 并运行它
[1] 10739
$
$ scp timeclnt.c bruce@computer2:           # 发送客户代码到另一台机器
bruce@computer2's password:
timeclnt.c        |  1KB       |  1.8KB/s      |ETA: 00:00:00 |100 %
$ ssh bruce@computer2
bruce@computer2's password:
No mail.
computer2:bruce$ cc timeclnt.c -o timeclnt
computer2:bruce $./timeclnt computer1 13000 Wow! Got a call!
The time here is  ..Tue Aug 14 02:44:31 2001
computer2:bruce$
```

服务器编译好后，运行在机器 1 上。然后把客户端程序复制到机器 2 上，再登录到机器 2。在机器 2 上，编译好客户端后，然后让客户端请求与运行在机器 1 上监听在 13 000 端口的服务器相连接。这里看到的消息就是从机器 1 上的服务器通过网络发送给机器 2 上的客户的。而客户再把消息发送至标准输出。

从上面的测试结果是否能真的看到机器 2 上的输出？我是从机器 1 连接到机器 2 的，所以显示消息的终端实际上是连接到机器 1 上的。后面的一些练习会让你仔细考虑其运作原理。

通过 timeserv/timeclnt 程序，可以得到另一台机器上的时间。检查另一台机器上的时间可以保证不同机器的时钟同步。网络上的某台机器可以作为权威时间服务器，而其他的机器可以利用上面的程序周期性地来调整自己的时钟。

11.5.8　另一种服务器：远程的 ls

下一个项目是编写一个可以打印远端机器上文件列表的程序。你可能在两台机器上拥有账号。若想看另一台机器上的文件列表时如何去做呢？可以登录到另一台机器上，然后运行 ls。一个快速的且更加方便的途径是运行一个远程的 ls 程序，这里称它为 rls(remote ls)。可以指定主机名和目录：

```
$ rls computer2.site.net /home/me/code
```

当然，rls 需要在另一台机器上的一个服务进程接收请求，处理请求和返回结果。该系统看上去类似于图 11.10。服务器运行在一台机器上，客户运行在另一台机器上并与服务器连接，把目录的名字发送给服务器。服务器将位于该目录下的文件列表信息返回给客户端。客户端将结果送至标准输出把它们显示出来。两进程系统提供了对于不同机器上的目录访问的支持。

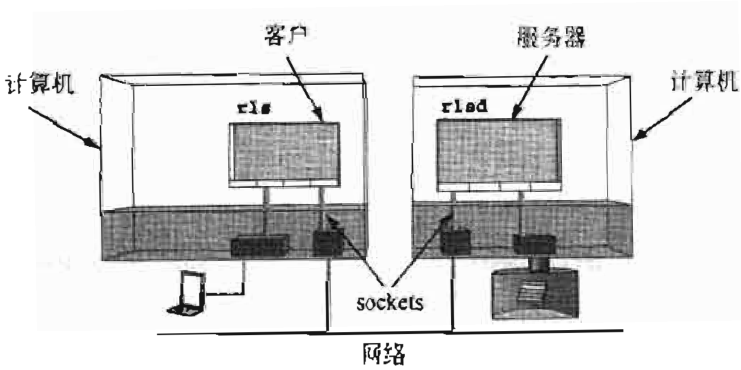


图 11.10　一个远端的 ls 系统

1. 设计远程 ls 系统

这里需要 3 个要素来实现 rls 系统：

(1) 协议

(2) 客户端程序

(3) 服务器端程序

2. 协议

协议包含有请求和应答。首先,客户端发送一行包含有目录名称的请求。服务器读取该目录名之后打开并读取该目录,然后把文件列表发送到客户端。客户端循环地读取文件列表,直到服务器挂断连接产生文件结尾标志。

3. 客户端程序:rls

```
/* rls.c - a client for a remote directory listing service
 *           usage: rls hostname directory
 */
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/socket.h>
#include   <netinet/in.h>
#include   <netdb.h>

#define    oops(msg)        { perror(msg); exit(1); }
#define    PORTNUM          15000

main(int ac, char *av[])
{
    struct sockaddr_in servadd;                 /* the number to call */
    struct hostent     *hp;                     /* used to get number */
    int    sock_id, sock_fd;                    /* the socket and fd */
    char   buffer[BUFSIZ];                      /* to receive message */
    int    n_read;                              /* for message length */

    if ( ac != 3 ) exit(1);

  /** Step 1: Get a socket **/

    sock_id = socket( AF_INET, SOCK_STREAM, 0 );  /* get a line  */
    if ( sock_id == -1 )
        oops( "socket" );                         /* or fail */

  /** Step 2: connect to server **/

    bzero( &servadd, sizeof(servadd) );           /* zero the address  */
    hp = gethostbyname( av[1] );                  /* lookup host's ip #  */
    if (hp == NULL)
        oops(av[1]);                              /* or die */
    bcopy(hp->h_addr, (struct sockaddr *)&servadd.sin_addr, hp->h_length);
    servadd.sin_port = htons(PORTNUM);            /* fill in port number */
    servadd.sin_family = AF_INET ;                /* fill in socket type */

    if ( connect(sock_id,(struct sockaddr *)&servadd, sizeof(servadd)) != 0)
        oops( "connect" );

/** Step 3: send directory name, then read back results **/
```

```
    if ( write(sock_id, av[2], strlen(av[2])) == -1)
        oops("write");
    if ( write(sock_id, "\n", 1) == -1 )
        oops("write");

    while( (n_read = read(sock_id, buffer, BUFSIZ)) > 0 )
        if ( write(1, buffer, n_read) == -1 )
                oops("write");
        close( sock_id );
}
```

注意该客户端与时间服务客户端的不同之处。rls 客户端首先把目录名写到 socket 中。上面的协议规定了客户端每次发送一行，因此程序中在行尾增加一个换行符。接下来，客户端进入一个循环，将从 socket 所接收的数据复制到标准输出，直到接收到文件结尾标志。rls.c 使用低级别的 write 和 read 调用来和服务器交换数据。循环中用到了标准大小的缓存以提高效率。下面将编写服务器端的程序。

4. 服务器程序：rlsd

服务器必须得到一个 socket，然后调用 bind、listen 命令，最后调用 accept 来接收一次呼叫。在接收呼叫之后，服务器从 socket 读取目录名，然后列出该目录下的内容。服务器是如何给出文件列表的呢？这里可以把第 3 章写的 ls 程序复制过来，但也可以用一个更简单的方法：仅仅使用 popen 读取常规版本的 ls 程序的输出，如图 11.11 所示。

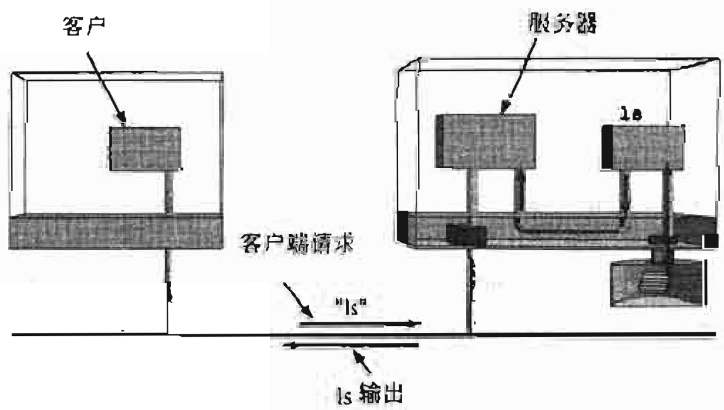


图 11.11 使用 popen"ls"来显示远端目录

下面的代码中使用了 popen：

```
/* rlsd.c - a remote ls server - with paranoia
 */
#include  <stdio.h>
#include  <unistd.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
```

```
#include   <netdb.h>
#include   <time.h>
#include   <strings.h>

#define  PORTNUM   15000                         /* our remote ls server port */
#define  HOSTLEN   256
#define  oops(msg)       { perror(msg) ; exit(1) ; }

int main(int ac, char * av[])
{
    struct   sockaddr_in    saddr;               /* build our address here */
    struct   hostent        * hp;                /* this is part of our */
    char     hostname[HOSTLEN];                  /* address */
    int      sock_id,sock_fd;                    /* line id, file desc */
    FILE     * sock_fpi, * sock_fpo;             /* streams for in and out */
    FILE     * pipe_fp;                          /* use popen to run ls */
    char     dirname[BUFSIZ];                    /* from client */
    char     command[BUFSIZ];                    /* for popen() */
    int      dirlen, c;

  /** Step 1: ask kernel for a socket **/

    sock_id = socket( PF_INET, SOCK_STREAM, 0 );    /* get a socket */
    if ( sock_id == -1 )
        oops( "socket" );

  /** Step 2: bind address to socket. Address is host,port **/

    bzero( (void *)&saddr, sizeof(saddr) );       /* clear out struct */
    gethostname( hostname, HOSTLEN );             /* where am I ? */
    hp = gethostbyname( hostname );               /* get info about host */
    bcopy( (void *)hp->h_addr, (void *)&saddr.sin_addr, hp->h_length);
    saddr.sin_port = htons(PORTNUM);              /* fill in socket port */
    saddr.sin_family = AF_INET ;                  /* fill in addr family */
    if ( bind(sock_id, (struct sockaddr *)&saddr, sizeof(saddr)) != 0 )
        oops( "bind" );

  /** Step 3: allow incoming calls with Qsize=1 on socket **/

    if ( listen(sock_id, 1) != 0 )
            oops( "listen" );

  /*
   * main loop: accept(), write(), close()
   */

   while ( 1 ){
       sock_fd = accept(sock_id, NULL, NULL);     /* wait for call */
       if ( sock_fd == -1 )
```

```
            oops("accept");
        /* open reading direction as buffered stream */
        if( (sock_fpi = fdopen(sock_fd,"r")) == NULL )
            oops("fdopen reading");

        if ( fgets(dirname, BUFSIZ-5, sock_fpi) == NULL )
            oops("reading dirname");
        sanitize(dirname);

        /* open writing direction as buffered stream */
        if ( (sock_fpo = fdopen(sock_fd,"w")) == NULL )
            oops("fdopen writing");

        sprintf(command,"ls %s", dirname);
        if ( (pipe_fp = popen(command, "r")) == NULL )
            oops("popen");

        /* transfer data from ls to socket */
        while( (c = getc(pipe_fp)) != EOF )
                putc(c, sock_fpo);
        pclose(pipe_fp);
        fclose(sock_fpo);
        fclose(sock_fpi);
    }
}

sanitize(char *str)
/*
 * it would be very bad if someone passed us an dirname like
 * "; rm *" and we naively created a command "ls ; rm *"
 *
 * so..we remove everything but slashes and alphanumerics
 * There are nicer solutions, see exercises
 */
{
    char *src, *dest;

    for ( src = dest = str ; *src ; src++ )
        if ( *src == '/' || isalnum(*src) )
                *dest++ = *src;
    *dest = '\0';
}
```

注意服务器程序使用标准缓存流来读写数据。服务器用 fgets 调用从客户端读取目录名。在调用 popen 后，服务器就像复制文件一样地使用 getc 和 putc 来传输数据。当然，服务器实际上是从本机上的进程向另一台机器上的进程复制数据。

注意 sanitize 函数的使用。对于任何运行参数中所含的命令或从因特网上获取数据的服务器，在编写的时候都要格外小心。程序中的服务器等待接收来自客户端的目录名，然后把它追加到 ls 命令的尾部。例如，如果客户端发送字符串"/bin"，服务器将创建并运行"ls /bin"命令。这是正确无误的。但是，如果有人发送字符串"; rm * "给服务器，服务器将创建并运行"ls ; rm * "命令了。

为了减少被破坏的风险，程序中必须确保接收的字符串没有溢出输入缓存，也没有溢出给命令设置的缓存并且接收的目录名中不允许出现非法字符。popen 系统调用对于编写网络服务来说是很危险的，因为它直接把一行字符串传给 shell。在网络程序中，将字符串传给 shell 是一个非常错误的想法。举这个例子的目的有两个：首先，展现 popen 的另一种用法；其次则是警告大家其危险性。在使用的时候务必小心！

11.6 软件精灵

像很多 Unix 程序一样，Unix 服务器程序有短小、简洁的名字。很多服务器程序都是以 d 结尾，如 httpd、inetd、syslogd 和 atd。这里的 d 表示精灵(daemon)的意思，因而如名叫 syslogd 的服务器程序实际上是系统日志精灵(system log daemon)。精灵就是一个为他人提供服务的帮助者，它随时等待去帮助别人。在你的系统中，输入命令 ps -el 或 ps -ax 就可以看到以字符 d 结尾的进程。然后，可以去阅读这些命令的帮助信息，从而可以更加深入地理解 Unix 中用客户/服务器模型是如何来处理一些基础操作的。

大部分精灵进程都是在系统启动后就处于运行状态了。位于类似于/etc/rc.d① 目录中的 shell 脚本在后台启动了这些服务，它们的运行与终端相分离，时刻准备提供数据或服务。

小　　结

1. 主要内容

- 一些程序被作为单独的进程建立起来来接收和发送数据。在客户/服务器模型中，服务器进程为客户进程提供处理或数据服务。
- 客户/服务器系统包含通信系统和协议。客户和服务器通过管道或 socket 进行通信。协议是会话过程中一系列规则的集合。
- popen 库函数可以将任何 shell 程序嵌入服务器程序并且让对服务器的访问就像访问缓存文件一样。
- 管道是一对相连接的文件描述符。socket 是一个未连接的通信端点，也是一个潜在的文件描述符。客户进程通过把自己的 socket 和服务器端的 socket 相连来创建一个通信连接。
- sockets 之间的连接可以扩展到另一台机器上。每个 socket 以机器地址和端口来标识。

① 准确的目录名依赖于 Unix 的版本。

- 到管道和 socket 的连接使用文件描述符。文件描述符为程序提供了与文件、设备和其他的进程通信的统一编程接口。

2. 下一步的工作

本章学习了客户/服务器模型编程的设计和两种连接进程的方法：管道和 socket。在下一章中，将集中精力学习客户/服务器模型编程的设计原理，并编写更加复杂的程序。特别地，下一章将把对 socket 编程和文件系统及进程控制的知识相结合来编写一个 Web 服务器程序。

3. 习题

11.1　如果你经营的是一家比萨饼外卖店而不是时间或查号辅助服务的话，结果将如何？协议将更加复杂。为送外卖服务描述在服务器和客户之间传送的消息序列。注意该协议包含有一个循环，以便允许客户增加定购项。

11.2　本章中的 popen 版本没有对信号做任何处理，这是不是正确？子进程从父进程那里继承了信号处理的设置。在考虑以下三种情况如何对父进程信号进行处理之后，回答该问题：终止、忽略以及调用函数。

11.3　在时间服务器和客户端运行的例子中，使用了 ssh 命令从机器 1 登录到机器 2。此时仍登录在机器 1 上，但是在机器 2 上却运行了一个 shell 进程。该 shell 程序编译并运行了时间服务客户端。
我的终端确实是连接到机器上了。重新画图 11.11，使得该图包含机器 1 上的 shell、机器 2 上的 shell、终端以及从 timeclnt 到终端正确的数据流。这可是一个相当复杂的数据流哦。

11.4　前面的章节中可以看到磁盘文件和设备文件都支持标准的文件接口，但是到磁盘文件的连接与到设备文件的连接具有完全不同的属性集。那么 socket 具有什么样的属性呢？参阅 setsockopt 的帮助手册以获取更详细的信息。

11.5　远端目录服务运行了 ls 命令。当 ls 发生错误的时候，将会发生什么事情呢？例如，指定的目录不存在或对于服务器而言不可读，那么对 ls 所产生的错误信息如何处理呢？可以考虑两种处理来自 ls 的错误信息的方法。首先，如何把错误信息发回给客户端？其次，如何把错误信息存放到日志中并通知用户？

4. 编程练习

11.6　给 tinybc 程序增加 -c 选项。一旦增加了该选项，下面的命令将能够工作：

```
printf "2 + 2\n4 * 4\n"|tinybc -c|dc
```

11.7　为你的 shell 增加 -c 选项。需要改变什么呢？

11.8　编写 pclose 程序。该函数以 popen 返回的 FILE * 作为参数。fdopen 函数为缓存和簿记信息分配了内存空间。fclose 函数释放该内存并关闭文件描述符。那么 pclose 还必须做些什么呢？若另一个子进程死在 popen 和 pclose 调用之间，

结果又将会怎样?

11.9　本章中的时间服务器并没有用到系统调用 accept 所提供的调用者 ID 的特性。修改 timeserv.c,使得它在接收到请求时可以打印出如 Got a call from 123.123.123.123 (computer2.mysite.net)。

可以阅读帮助手册和头文件来了解该工程中所需要的函数和结构体。

11.10　写一个程序将 sort 作为子程序调用。程序须读取多行数据,存放到字符串数组中。接着程序创建两个管道,然后创建一个进程来运行 sort。通过一个管道把输入序列发送给 sort 的输入,再关闭该管道。通过另一个管道读取 sort 的输出,再把结果存回数组中,并打印该数组。

11.11　基于 System V 的 Unix 版本提供了对双向管道的支持。通过运行下面的程序,可以测试某个版本的 Unix 是否支持双向管道:

```
/*
 * testbpd.c - test bidirectional pipes
 */

main()
{
    int p[2];

    if ( pipe(p) == -1 ) exit(1);
    if ( write(p[0], "hello", 5) == -1 )
        perror("write into pipe[0] failed");
    else
        printf("write into pipe[0] worked\n");
}
```

在内部,双向管道含有两个队列,一个从 pipe[0]到 pipe[1],另一个则方向相反。向管道的一端写数据操作会把数据放入到通向另一端的队列中,而从管道的一端读数据则将数据从那端取出并送入本端的队列中。

如果你的系统不支持双向管道,可以生成如下调用:

```
#include <sys/types.h>
#include <sys/socket.h>
int apipe[2];      /* a pipe */
socketpair(AF_UNIX, SOCK_STREAM, PF_UNSPEC, apipe);
```

重新编制程序 tinybc.c,使它使用一个双向管道而不是使用两个单向的管道。

11.12　更改 timeserv.c 程序,使得它只响应来自特定 IP 地址主机的客户端。服务器接收呼叫并检查客户端地址。如果客户端地址不是特定的 IP,则服务器挂断,否则,服务器将时间返回。

增强该阻塞特征使服务器可以从文件中读取所能接收的 IP 地址列表。描述该

技术的一些实际应用。

11.13 大家知道在服务器端使用 popen 调用是非常危险的。有两种方法来解决该问题。第一种方法是编写一个更加灵活，但非常安全的 sanitize 函数。例如，目录名中允许含有逗号、破折号、空格和其他字符。并且目录名中还可以含有星号和分号，这个处理可以像 shell 一样，给这些字符赋予特定的含义。写一个更加有用的，但是安全的字符串解析函数。

另外一种方法是放弃使用 popen，用 fork、exec 和 dup 等函数。用这种方法来重写 rlsd.c 程序。其中需要用到 wait 吗？为什么？

11.14 编写一个位于端口 79 的目录辅助服务器(finger 服务器)。服务器接收单行的用户名输入，然后给客户端发送一个列表，其中包含与输入匹配的所有用户。

11.15 代理(proxy)指的是接收请求，把该请求转发给其他的服务器，然后再将从服务器中传回的结果返回给程序。这就类似于干洗店的前台：它不做清洁工作，只是把衣服传送到洗衣设备或从洗衣设备取衣服。

编写一个时间服务器代理。你的程序应当可以接收标准端口上的连接。为了处理该连接，程序需要和真的时间服务建立连接，从该服务器取得时间，然后把时间转发给客户端。

11.16 考虑上一题中的关于代理服务器的概念。时间只是每秒钟改变一次，如果你的代理服务器在几毫秒中接收了很多请求，就根本不可能在这么短的时间内向时间服务器发送许多请求。编写一个时间代理服务器，可以缓存从时间服务器读取的时间，只有在新的呼叫超过了 1 秒的间隔之后才去向时间服务器发送请求(参见 gettimeofday)。

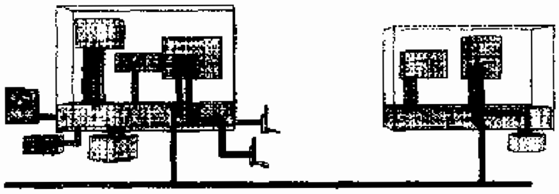
11.17 必须承认，在上一题中使用带缓存的时间服务是一个愚蠢的想法。但在 finger 服务器中采用缓存技术则是有意义，解释原因。编写一个目录辅助服务器，使其可以缓存用户信息。

缓存时间服务器中所缓存的每个元素都有自然的生命周期(1 秒钟)，但是在缓存的目录辅助服务器中如何决定用户信息的保存时间呢？

11.18 有些面包店有一台给客户分发编号的机器。柜台上写着“正在服务”的牌子上显示了下一个顾客的编号。设计一个客户/服务器程序来实现面包数字服务器系统。服务器产生连续编号。用户运行客户端程序来获取服务器端的数字。

11.19 每个 C 程序员都知道 argv[0]通常表示正在运行的程序名称。有一种比较接近的方法可以使一个进程获得自己的名字。程序可以使用 popen，然后从 ps 命令的输出中来搜索自己的进程 ID。编写一个使用该方法的程序。

第 12 章　连接和协议：编写 Web 服务器



概念与技巧

- 服务器端 socket：目的和构造
- 客户端 socket：目的和构造
- 客户/服务器协议
- 服务器设计：使用 fork 来接收多个请求
- 僵尸(zombie)问题
- HTTP

12.1　服务器设计重点

使用万维网是容易的，只要在浏览器中输入一个网址或者单击一个超链，服务器就会把相应的网页发送过来。但是 Web 是怎样工作的？从 Web 服务器上获取网页和从时间服务器获取时间是类似的吗？

基于 socket 的客户/服务器系统大多是类似的。虽然电子邮件、文件传输、远程登录和分布式数据库，以及其他的 Internet 服务在屏幕上显示的内容相异，但是它们的运作原理是一致的。

一旦理解了一个 socket 流的客户/服务器系统，就可以理解大多数其他的系统。在本章中，将学习网络编程的基本操作和设计原则，然后使用它们来建立一个 Web 服务器。

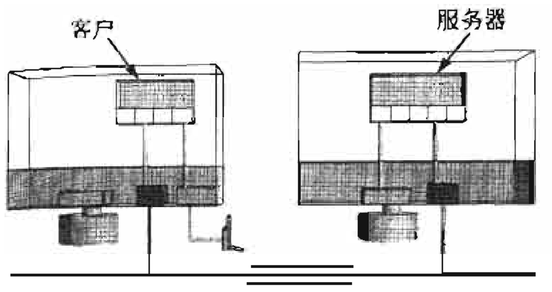
12.2　三个主要操作

在第 11 章看到的基于 socket 流的客户/服务器系统与图 12.1 看上去类似。客户和服务器都是进程。服务器设立服务，然后进入循环接收和处理请求。客户连接到服务器，然后发送、接受或者交换数据，最后退出。该交互过程中主要包含了以下 3 个操作：

(1) 服务器设立服务。

(2) 客户连接到服务器。

(3) 服务器和客户处理事务。



客户：		服务器：
		设立服务
连接到服务器	---------------→	接收请求 ←
获取服务	←—————————	提供服务 │
挂断连接		挂断连接 ——┘

图 12.1　客户/服务器交互中的主要步骤

下面将分别讨论每个操作。

12.3　操作 1 和操作 2：建立连接

基于流的系统需要建立连接。这里将回顾一下建立连接的步骤，然后将这些步骤抽象成一些库函数。

12.3.1　操作 1：建立服务器端 socket

首先，如图 12.2 所示，描述了服务器设立一个服务的过程。设立一个服务一般需要如下 3 个步骤：

（1）创建一个 socket

```
socket = socket(PF_INET, SOCK_STREAM,0)
```

（2）给 socket 绑定一个地址

```
bind(sock, &addr, sizeof(addr))
```

（3）监听接入请求

```
listen(sock, queue_size)
```

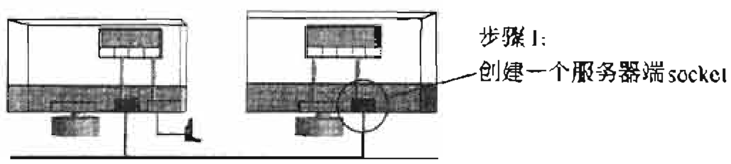


图 12.2　创建服务器端 socket

为了避免在编写服务器时重复输入上述代码，将这 3 个步骤组合成一个函数：make_server_socket。该函数的代码位于本章后面将给出的 socklib.c 文件中。在编写服务器的时候，只要调用该函数就可以创建一个服务器端 socket。具体如下：

```
sock = make_server_socket(int portnum)

return -1 if error,
       or a server socket listening at port "portnum"
```

12.3.2 操作 2：建立到服务器的连接

其次，将客户连接到服务器。如图 12.3 所示，基于流的网络客户连接到服务器包含以下两个步骤：

（1）创建一个 socket

```
socket = socket(PF_INET, SOCK_STREAM, 0)
```

（2）使用该 socket 连接到服务器

```
connect(sock, &serv_addr, sizeof(serv_addr))
```

将这两个步骤抽象成一个函数：connect_to_server。当编写客户端程序时，只要调用该函数就可以建立到服务器的连接。具体如下：

```
fd = connect_to_server(hostname, portnum)

return -1 if error,
or a fd open for reading and writing connected to the socket at port "portnum" on host "hostname"
```

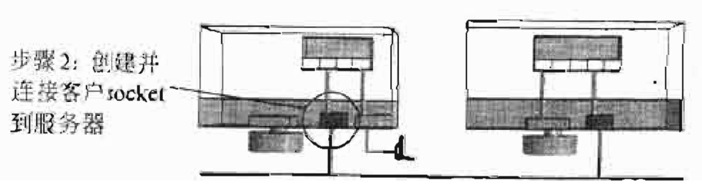


图 12.3 连接到服务器

12.3.3 socklib.c

```
/* socklib.c
 *
 * This file contains functions used lots when writing internet
 * client/server programs. The two main functions here are:
 *
 * int make_server_socket( portnum ) returns a server socket
 *                                   or -1 if error
 * int make_server_socket_q(portnum,backlog)
 *
 * int connect_to_server(char * hostname, int portnum)
```

```
 *                              returns a connected socket
 *                              or  1 if error
 */
#include  <stdio.h>
#include  <unistd.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <netdb.h>
#include  <time.h>
#include  <strings.h>

#define   HOSTLEN  256
#define   BACKLOG  1

int make_server_socket_q(int , int );

int make_server_socket(int portnum)
{
    return make_server_socket_q(portnum, BACKLOG);
}
int make_server_socket_q(int portnum, int backlog)
{
    struct sockaddr_in saddr;                       /* build our address here */
    struct hostent *hp;                             /* this is part of our */
    char   hostname[HOSTLEN];                       /* address */
    int    sock_id;                                 /* the socket */

    sock_id = socket(PF_INET, SOCK_STREAM, 0);      /* get a socket */
    if ( sock_id == -1 )
        return -1;

    /* * build address and bind it to socket * */

    bzero((void *)&saddr, sizeof(saddr));           /* clear out struct */
    gethostname(hostname, HOSTLEN);                 /* where am I ? */
    hp = gethostbyname(hostname);                   /* get info about host */
                                                    /* fill in host part */
    bcopy( (void *)hp->h_addr, (void *)&saddr.sin_addr,
        hp->h_length);
    saddr.sin_port = htons(portnum);                /* fill in socket port */
    saddr.sin_family = AF_INET ;                    /* fill in addr family */
    if ( bind(sock_id, (struct sockaddr *)&saddr,
        sizeof(saddr)) != 0 )
    return -1;

    /* * arrange for incoming calls * */
```

```
    if ( listen(sock_id, backlog) != 0 )
    return -1;
    return sock_id;
}

int connect_to_server(char * host, int portnum)
{
    int sock;
    struct sockaddr_in servadd;                    /* the number to call */
    struct hostent      * hp;                      /* used to get number */

    /* * Step 1: Get a socket * */

    sock = socket( AF_INET, SOCK_STREAM, 0 );      /* get a line */
    if ( sock == -1 )
        return -1;

    /* * Step 2: connect to server * */

    bzero( &servadd, sizeof(servadd) );            /* zero the address */
    hp = gethostbyname( host );                    /* lookup host's ip # */
    if (hp == NULL)
        return -1;
    bcopy(hp->h_addr, (struct sockaddr *)&servadd.sin_addr,
        hp->h_length);
    servadd.sin_port = htons(portnum);             /* fill in port number */
    servadd.sin_family = AF_INET ;                 /* fill in socket type */

    if ( connect(sock,(struct sockaddr *)&servadd,
        sizeof(servadd)) != 0)
       return -1;
    return sock;
}
```

12.4　操作 3：客户/服务器的会话

至此，可以使用专门的函数来建立服务器端的 socket，同时也有专门的函数来连接到服务器。在实践中，如何利用上述函数呢？客户和服务器之间的交互内容又是什么呢？在本节中，将学习客户端程序和服务器端程序编写的一般形式，以及一些建立服务器的设计方案。

1. 一般的客户端

网络客户通常调用服务器来获得服务，一个典型的客户程序如下：

```
main()
{
    int fd;
    fd = connect_to_server(host,port);    /* call the server */
```

```
    if(fd == -1)
        exit(1);                        /* or die */
    talk_with_server(fd);               /* chat with server */
    close(fd);                          /* hang up when done */
}
```

函数 talk_with_server 处理与服务器的会话。具体的内容取决于特定应用。例如，e-mail客户和邮件服务器交谈的是邮件，而天气预报客户和服务器交谈的则是天气。

2. 一般的服务器端

一个典型的服务器如下：

```
main()
{
    int sock,fd;                          /* socket and connection */
    sock = make_server_socket(port);
    if(sock == -1)
        exit(1);
    while(1)
    {
        fd = accept(sock,NULL,NULL);      /* take next call */
        if(fd == -1)
            break;                        /* or die */
    process_request(fd);                  /* chat with client */
    close(fd);                            /* hang up when done */
    }
}
```

函数 process_request 处理客户的请求。具体的内容取决于特定应用。例如，邮件服务器告诉客户信件信息，天气服务器则告诉客户天气情况。

12.4.1 使用 socklib.c 的 timeserv/timeclnt

如何利用上面的模板来建立客户/服务器系统呢？例如，在该框架下本文的时间系统客户/服务器是怎样的呢？图 12.4 对此做了解释。为了使用 socklib.c 重写时间客户和服务器，这里编写了处理会话的函数 talk_with_server 用于客户端，而 process_request 用于服务

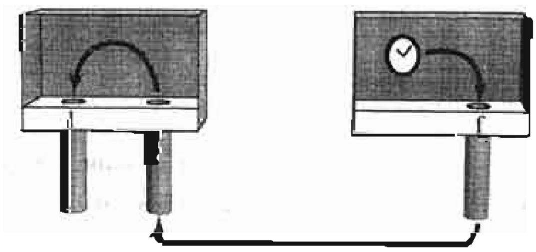


图 12.4 时间服务器和客户端版本 1

器端。

```
talk_with_server(fd)
{
  char buf[LEN];
  int n;

  n = read(fd,buf,LEN);
  write(1,buf,n);
}
```

```
process_request(fd)
{
  time_t now;
  char *cp;
  time (&now);
  cp = ctime(&now);
  write(fd,cp,strlen(cp));
}
```

服务器调用 time 从内核中获得时间，然后用 ctime 将时间转换成可以打印的字符串。服务器将该字符串写到 socket 中，发送给客户端的 socket。客户从 socket 中读取该字符串，然后写到标准输出中。这个新的版本遵循了先前版本的程序逻辑，但是设计更加模块化，代码更加清晰。

12.4.2 第 2 版的服务器：使用 fork

现在考虑第二版服务器的设计。第二版中程序没有通过调用 time 函数来获得代表时间的数据，而是直接使用了一个 shell 命令（date 命令），如图 12.5 所示。

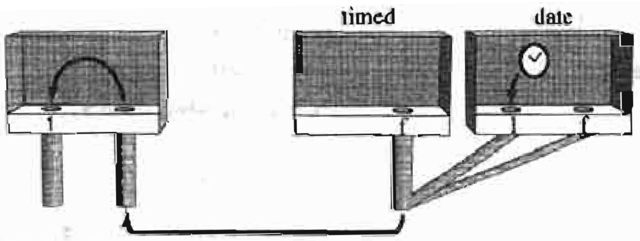


图 12.5 服务器使用 fork 运行 date

代码如下：

```
process_request(fd)
/*
 * send the date out to the client via fd
 */
{
    int pid = fork();
    switch(pid)
    {
        case -1:return;                  /* can not provide service */
        case 0:dup2(fd,1);               /* child runs date */
               close(fd);                /* by redirecting stdout */
               execl("/bin/date","date",NULL);
               oops("execlp");           /* or quits */
```

```
        default: wait(NULL);            /* parent wait for child */
    }
}
```

如图 12.5 所示，服务器用 fork 建立一个新的子进程。该子进程将标准输出重定向到 socket，然后运行 date。date 命令给出日期，然后将日期写到标准输出，这样就把字符串发送到客户端了。在程序中调用了 wait。shell 通常在调用 fork 后要调用 wait，那么这里的调用有意义吗？本文将在下一节中探讨该问题。

12.4.3　服务器的设计问题：DIY 或代理

这里使用了两种服务器的设计方法：

- 自己做(Do It Yourself，DIY)——服务器接收请求，自己处理工作。
- 代理——服务器接收请求，然后创建一个新进程来处理工作。

每种方法的优缺点各是什么？

- 自己做用于快速简单的任务

计算当前的日期和时间需要系统调用 time 和库函数 ctime。使用 fork 和 exec 来运行 date 至少需要 3 个系统调用和创建一个新的进程。对于一些服务器，效率最高的方法是服务器自己来完成工作并且在 listen 中限制连接队列的大小。文件 socklib.c 中的 make_server_socket_q 函数以队列大小作为参数。

- 代理用于慢速的更加复杂的任务

服务器处理耗时的任务或等待资源时，需要代理来完成其工作。这就像商务中的电话接线员，接收电话，把连接传递到下一个销售或服务人员，然后再回过去接收下一个电话，而服务器可以使用 fork 创建一个新进程来处理每个请求。通过这种方式，服务器可以同时处理多个任务。

- 使用 SIGCHLD 来阻止僵尸(zombie)问题

除了等待子进程死亡外，父进程可以设置为接收表示子进程死亡的信号。第 8 章中解释了当子进程退出或被终止时，内核发送 SIGCHLD 给父进程。但它不同于本文讨论的其他信号，默认时 SIGCHLD 是被忽略的。父进程可以为 SIGCHLD 设置一个信号处理函数，它可以调用 wait。具体方法如下：

```
/* naive use of SIGCHLD handler with wait() - buggy */
main()
{
int sock, fd;
void child_waiter(int),process_request(int);
signal(SIGCHLD,child_waiter);
if((sock = make_server_socket(PORTNUM)) = = - 1)
    oops("make_server_socket");
}
while(1) {
    fd = accept(sock,NULL,NULL);
```

```
        if(fd == -1)
          break;
      process_request(fd);
      close(fd);
        }
    }
    void child_waiter(int signum)
    {
        wait(NULL);
    }
    void process_request(int fd)
    {
        if(fork() == 0) {                      /* child */
          dup2(fd,1);                          /* moves socket to fd 1 */
          close(fd);                           /* closes socket */
          execlp("data","date",NULL);          /* exec date */
          oops("execlp date");
        }
    }
```

下面来分析程序中的流程控制。当一个请求过来时，父进程使用 fork，然后父进程立即返回去接收下一个请求，让子进程去处理请求。当子进程退出时，父进程收到 SIGCHLD 信号，跳到处理函数并调用 wait。子进程从进程表中被删除，父进程从处理函数返回到主函数。该过程看上去似乎很完美了，不过其中存在着两个问题。

问题 1　是程序运行到信号处理函数跳转时会中断系统调用 accept。当 accept 被信号中断时，返回−1，然后设置 errno 到 EINTR。代码中把 accept 返回的−1 作为错误，然后从主循环中跳出来。这里需要更改 main 函数来区分真正的错误和被打断的系统调用所产生的错误。这个作为练习，由读者完成。

问题 2　关于 Unix 是如何处理多个信号的。如果多个子进程几乎同时退出，将会发生什么？假设同时有 3 个 SIGCHLD 发送到父进程。最先到达的信号导致父进程跳到处理函数，然后父进程调用 wait 来保证子进程已经从进程表中删除。这样就可以了吗？

当父进程在运行信号处理函数时，其他两个信号的到达导致 Unix 阻塞，但是并不缓存信号。从而，第二个信号被阻塞，而第三个信号丢失了。此时，如果还有其他的子进程退出，来自于这些子进程的信号也将丢失。信号处理函数只调用了 wait 一次，所以每次丢失一个信号意味着少调用了一次 wait，这将产生更多的僵尸进程(zombie)。解决方法是在处理函数中调用 wait 足够多的次数来去除所有的终止进程。waitpid 函数解决了此问题：

```
void child_waiter(int signum)
{
while(waitpid(-1,NULL,WNOHANG)>0);
}
```

waitpid 提供了 wait 函数超集的功能。其第一个参数表示它所要等待的进程 ID 号。值 -1 表示等待所有的子进程。第二个参数是指向整型值的指针，用来获取状态。服务器并不关心子进程中发生了什么，不过一个健壮的服务器可能用该信息来跟踪错误。

waitpid 的最后一个参数表示选项。WNOHANG 参数告诉 waitpid：如果没有僵尸进程，则不必等待。

该循环直到所有退出的子进程都被等待了才停止。即使多个子进程同时退出并产生了多个 SIGCHLD，所有的这些信号都会被处理。

12.5　编写 Web 服务器

至此，已经学习了编写 Web 服务器的必备知识。Web 服务器是已经编写的目录服务器的扩展。主要的扩展是一个 cat 服务器和一个 exec 服务器。

12.5.1　Web 服务器功能

Web 服务器通常要具备 3 种用户操作：

(1) 列举目录信息。

(2) cat 文件。

(3) 运行程序。

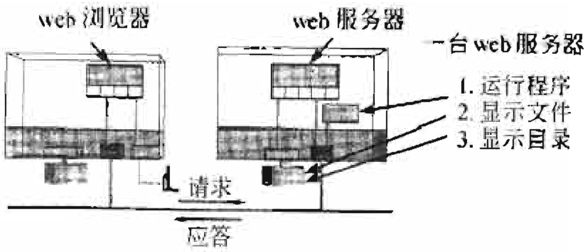


图 12.6　Web 服务器提供远程 ls、cat、exec

Web 服务器通过基于流的 socket 连接为客户提供上述 3 种操作。如图 12.6 所示，用户连接到服务器后，发送请求，然后服务器返回客户请求的信息。具体过程如下：

客户端：		服务器端：
用户选择一个链接		
连接服务器	→	接收请求
写请求	→	读取请求
		处理请求：
		目录：显示目录列表
		文件：显示内容
		.cgi 文件：运行
		不存在：错误消息
读取应答	←	写应答

挂断
显示应答
　　html:解析
　　image:绘图
　　sound:运行
重复

12.5.2 设计 Web 服务器

所要编写的操作如下。

(1) 建立服务器

可以使用 socklib.c 中的 make_server_socket。

(2) 接收请求

使用 accept 来得到指向客户端的文件描述符。可以使用 fdopen 使得该文件描述符转换成缓冲流。

(3) 读取请求

什么是一个请求? 客户端如何请求服务? 这些需要进一步学习。

(4) 处理请求

已经知道了如何列出目录信息、cat 文件以及运行程序。通过 opendir 和 readdir、open 和 read、dup2 和 exec 的使用可以实现上述功能。

(5) 发送应答

什么是一个应答? 客户端期待接收的又是什么? 这些也需要进一步地学习。至此,已经学习了几乎所有的编写 Web 服务器的知识和技能,所剩的是 Web 服务器协议的学习。

12.5.3 Web 服务器协议

客户端(浏览器)与 Web 服务器之间的交互主要包含客户的请求和服务器的应答。请求和应答的格式在超文本传输协议(HTTP)中有定义。HTTP 像上一章中的时间服务器和 finger 服务器的协议一样,使用纯文本。就像在时间服务器和 finger 服务器中所做的,这里可以使用 telnet 和 Web 服务器进行交互。Web 服务器在端口 80 监听。下面是一个实际的例子:

```
$ telnet www.prenhall.com 80
Trying 165.193.123.253...
Connected to www.prenhall.com.
Escape character is '^]'.
GET /index.html HTTP/1.0

HTTP/1.1 200 OK
Server:Netscape-Enterprise/3.6 SP3
Date:Tue,22 Jan 2002 16:11:14 GMT
Content-type:text/html
```

```
Last - modified:Fri, 08 Sep 2000 20:20:06 GMT
Content - length:327
Accept - ranges:bytes
Connection:close

<HTML><HEAD>
<META HTTP-EQUIV = "Refresh"CONTENT = "0;
URL = http://vig.prenhall.com/">
</HEAD><BODY></BODY></HTML>
<! -- ------------------------------------------------------------------ >
<! --                     Caught you peeking!                          -- >
<! -- ------------------------------------------------------------------ >
Connection closed by foreign host.
$
```

这里只发送了一行请求，却接收了多行返回。知道具体细节吗？

1. HTTP 请求：GET

telnet 创建了一个 socket 并调用了 connect 来连接到 Web 服务器。服务器接受连接请求，并创建了一个基于 socket 的从客户端的键盘到 Web 服务进程的数据通道。

接下来，输入请求：

```
GET /index.html HTTP/1.0
```

一个 HTTP 请求包含有 3 个字符串。第一个字符串是命令，第二个是参数，第三个是所用协议的版本号。在该例子中，使用了 GET 命令，以 index.html 作为参数，使用了 HTTP 版本 1.0。

HTTP 还包含几个其他的命令。大部分 Web 请求使用 GET，因为大部分时间中用户是单击链接来获取网页。GET 命令可以跟几行参数。这里使用了简单的请求，以一个空行来表示参数的结束，并使用与本书前面提及的关于 shell 的相同约定。实际上，一个 Web 服务器只是集成了 *cat* 和 ls 的 Unix shell。

2. HTTP 应答：OK

服务器读取请求，检查请求，然后返回一个请求。应答有两部分：头部和内容。头部以状态行起始，如下所示：

```
HTTP/1.1 200 OK
```

状态行含有两个或更多的字符串。第一个串是协议的版本，第二个串是返回码，这里是 200，其文本的解释是 OK。这里请求的文件叫/info.html，而服务器给出应答表示可以得到该文件。如果服务器中没有所请求的文件名，返回码将是 404，其解释将是“未找到”。

头部的其余部分是关于应答的附加信息。在该例子中，附加信息包含服务器名、应答时间、服务器所发送数据类型以及应答的连接类型。一个应答头部可以包含有多行信息，以空行表示结束，空行位于 Connection:close 后面。

应答的其余部分是返回的具体内容。这里，服务器返回了文件/index.html 的内容。

3. HTTP 小结

客户端和 web 服务器交互的基本结构如下：

(1) 客户发送请求

GET filename HTTP/version

可选参数

空行

(2) 服务器发送应答

HTTP/version status-code status-message

附加信息

空行

内容

协议的完整描述可以参阅网上的版本 1.0 的 RFC1945 和版本 1.1 的 RFC2068。

Web 服务器必须接收客户的 HTTP 请求，并发送 HTTP 应答。请求和应答采用纯文本格式，是为了便于使用 C 中的输入/输出以及字符串函数读取和处理。

12.5.4 编写 Web 服务器

要求 Web 服务器只支持 GET 命令，只接收请求行，跳过其余参数，然后处理请求和发送应答，主要循环如下：

```
while(1)
{
    fd = accept(sock,NULL,NULL);          /* take a call */
    fpin = fdopen(fd,"r");                /* make it a FILE * */
    fgets(fpin, request,LEN);             /* read client request */
    read_until_crnl(fpin);                /* skip over arguments */
    process_rq(request,fd);               /* reply to client */
    fclose(fpin);                         /* hang up connection */
}
```

为了简洁起见，这里忽略了出错检查。

1. 处理请求

处理请求包含识别命令和根据参数进行处理：

```
process_rq( char * rq, int fd )
{
    char cmd[11], arg[513];

    if ( fork() != 0 )                        /* if a child, do work */
            return;                           /* if parent,return */

    sscanf(rq, "%10s%512s", cmd, arg);
        if ( strcmp(cmd,"GET") != 0 )         /* check command */
```

```
            cannot_do(fd);
    else if ( not_exist( arg ) )                /* does the arg exist */
            do_404(arg, fd );                   /* n: tell the user */
    else if ( isadir( arg ) )                   /* is it a directory? */
            do_ls( arg, fd );                   /* y: list contents */
    else if ( ends_in_cgi( arg ) )              /* name is X.cgi? */
            do_exec( arg, fd );                 /* y: execute it */
    else                                        /* otherwise */
            do_cat( arg, fd );                  /* display contents */
}
```

服务器为每个请求创建一个新的进程来处理。子进程将请求分割成命令和参数。如果命令不是 GET，服务器应答 HTTP 返回码表示未实现的命令。如果命令是 GET，服务器将期望得到目录名，一个以 .cgi 结尾的可执行程序或文件名。如果没有该目录或指定的文件名，服务器报错。

如果存在目录或文件，服务器决定所要使用的操作：ls、exec 或 cat。

2. 目录列表函数

函数 do_ls 处理列出目录信息的请求：

```
do_ls(char *dir, int fd)
{
    FILE *fp ;

    fp = fdopen(fd,"w");                        /* make socket into a FILE* */
    header(fp, "text/plain");                   /* send HTTP reply header */
    fprintf(fp,"\r\n");                         /* and end of header mark */
    fflush(fp);                                 /* force to socket */

    dup2(fd,1);                                 /* make socket stdout */
    dup2(fd,2);                                 /* make socket stderr */
    close(fd);                                  /* close socket */
    execl("/bin/ls","ls","-l",dir,NULL);        /* ls -l does the work */
    perror(dir);                                /* or it doesn't */
    exit(1);                                    /* child exits */
}
```

这里没有像前面章节中的目录服务一样使用 popen，而是通过调用 ls 命令，避免了客户向 shell popen 传递任意字符串来运行的问题。

3. 其他函数

其他的函数包含在本章的后面部分中。程序可以工作，但它并不完整，也不安全。需要做如下的改进：

（1）僵尸进程的去除；

（2）缓存溢出保护；

(3) CGI(Common Gateway Interface,通用网关接口)程序需要设置一些环境变量;

(4) HTTP 头部可以包含更多的信息。

该程序是一个包含 230 行 C 代码的完整的 Web 服务器,包含注释和空行。

12.5.5 运行 Web 服务器

编译程序,在某个端口运行它:

```
$ cc webserv.c socklib.c -o webserv
$ ./webserv 12345
```

现在可以访问 Web 服务器,网址为 http://yourhostname:12345/。将 html 文件放到该目录中并且用 http://yourhostname:12345//filename.html 来打开它。创建下面的 shell 脚本:

```
#!/bin/sh
# hello.cgi-a cheery cgi page
printf "Content-type:text/plain\n\nhello\n";
```

将它命名为 hello.cgi,用 chmod 改变权限为 755,然后用浏览器调用该程序:http://yourhostname:12345/hello.cgi。

12.5.6 Webserv 的源程序

下面是简单 Web 服务器的代码:

```
/* webserv.c    a minimal web server (version 0.2)
 *      usage: ws portnumber
 *    features: supports the GET command only
 *              runs in the current directory
 *              forks a new child to handle each request
 *              has MAJOR security holes, for demo purposes only
 *              has many other weaknesses, but is a good start
 *      build: cc webserv.c socklib.c -o webserv
 */
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/stat.h>
#include   <string.h>

main(int ac, char *av[])
{
    int     sock, fd;
    FILE    *fpin;
    char    request[BUFSIZ];
```

```
    if ( ac == 1 ){
        fprintf(stderr,"usage: ws portnum\n");
        exit(1);
    }
    sock = make_server_socket( atoi(av[1]) );
    if ( sock == -1 ) exit(2);

    /* main loop here */

    while(1){
        /* take a call and buffer it */
        fd = accept( sock, NULL, NULL );
        fpin = fdopen(fd, "r" );

        /* read request */
        fgets(request,BUFSIZ,fpin);
        printf("got a call: request = %s", request);
        read_til_crnl(fpin);

        /* do what client asks */
        process_rq(request, fd);

        fclose(fpin);
    }
}

/* ------------------------------------------------------------------------ *
   read_til_crnl(FILE *)
   skip over all request info until a CRNL is seen
   ------------------------------------------------------------------------ */

read_til_crnl(FILE *fp)
{
    char buf[BUFSIZ];
    while( fgets(buf,BUFSIZ,fp) != NULL && strcmp(buf,"\r\n") != 0 );
}

/* ------------------------------------------------------------------------ *
   process_rq( char *rq, int fd )
   do what the request asks for and write reply to fd
   handles request in a new process
   rq is HTTP command: GET /foo/bar.html HTTP/1.0
   ------------------------------------------------------------------------ */

process_rq( char *rq, int fd )
{
    char    cmd[BUFSIZ], arg[BUFSIZ];

    /* create a new process and return if not the child */
```

```
        if ( fork() != 0 )
                return;

        strcpy(arg, "./");                  /* precede args with ./ */
        if ( sscanf(rq, "%s%s", cmd, arg+2) != 2 )
                return;

        if ( strcmp(cmd,"GET") != 0 )
                cannot_do(fd);
        else if ( not_exist( arg ) )
                do_404(arg, fd );
        else if ( isadir( arg ) )
                do_ls( arg, fd );
        else if ( ends_in_cgi( arg ) )
                do_exec( arg, fd );
        else
                do_cat( arg, fd );
}

/* ------------------------------------------------------------------------ *
   the reply header thing: all functions need one
   if content_type is NULL then don't send content type
   ------------------------------------------------------------------------ */

header( FILE *fp, char *content_type )
{
   fprintf(fp, "HTTP/1.0 200 OK\r\n");
   if ( content_type )
      fprintf(fp, "Content-type: %s\r\n", content_type );
   }

/* ------------------------------------------------------------------------ *
   simple functions first:
      cannot_do(fd)       unimplemented HTTP command
    and do_404(item,fd)    no such object
   ------------------------------------------------------------------------ */

cannot_do(int fd)
{
    FILE *fp = fdopen(fd,"w");

    fprintf(fp, "HTTP/1.0 501 Not Implemented\r\n");
    fprintf(fp, "Content-type: text/plain\r\n");
    fprintf(fp, "\r\n");

    fprintf(fp, "That command is not yet implemented\r\n");
    fclose(fp);
}
```

```
do_404(char * item, int fd)
{
    FILE * fp = fdopen(fd,"w");

    fprintf(fp, "HTTP/1.0 404 Not Found\r\n");
    fprintf(fp, "Content - type: text/plain\r\n");
    fprintf(fp, "\r\n");

    fprintf(fp, "The item you requested: % s\r\nis not found\r\n", item);
    fclose(fp);
}

/* ------------------------------------------------------------------ *
   the directory listing section
   isadir() uses stat, not_exist() uses stat
   do_ls runs ls. It should not
   ------------------------------------------------------------------ */

isadir(char * f)
{
    struct stat info;
    return ( stat(f, &info) != - 1 && S_ISDIR(info.st_mode) );
}

not_exist(char * f)
{
    struct stat info;
    return( stat(f,&info) == - 1 );
}

do_ls(char * dir, int fd)
{
    FILE * fp ;

    fp = fdopen(fd,"w");
    header(fp, "text/plain");
    fprintf(fp,"\r\n");
    fflush(fp);

    dup2(fd,1);
    dup2(fd,2);
    close(fd);
    execlp("ls","ls"," l",dir,NULL);
    perror(dir);
    exit(1);
}

/* ------------------------------------------------------------------ *
```

```
   the cgi stuff. function to check extension and
   one to run the program.
   ------------------------------------------------------------------ */

char * file_type(char * f)
/* returns 'extension' of file */
{
    char    * cp;
    if ( (cp = strrchr(f, '.' )) != NULL )
            return cp + 1;
    return "";
}

ends_in_cgi(char * f)
{
    return ( strcmp( file_type(f), "cgi" ) == 0 );
}

do_exec( char * prog, int fd )
{
    FILE * fp ;

    fp = fdopen(fd,"w");
    header(fp, NULL);
    fflush(fp);
    dup2(fd, 1);
    dup2(fd, 2);
    close(fd);
    execl(prog,prog,NULL);
    perror(prog);
}

/* ------------------------------------------------------------------ *
   do_cat(filename,fd)
   sends back contents after a header
   ------------------------------------------------------------------ */

do_cat(char * f, int fd)
{
    char    * extension = file_type(f);
    char    * content = "text/plain";
    FILE    * fpsock, * fpfile;
    int     c;

    if ( strcmp(extension,"html") == 0 )
            content = "text/html";
    else if ( strcmp(extension, "gif") == 0 )
```

```
            content = "image/gif";
    else if ( strcmp(extension, "jpg") == 0 )
            content = "image/jpeg";
    else if ( strcmp(extension, "jpeg") == 0 )
            content = "image/jpeg";

    fpsock = fdopen(fd, "w");
    fpfile = fopen( f , "r");
    if ( fpsock != NULL && fpfile != NULL )
    {
        header( fpsock, content);
        fprintf(fpsock, "\r\n");
        while( (c = getc(fpfile) ) != EOF )
            putc(c, fpsock);
        fclose(fpfile);
        fclose(fpsock);
    }
    exit(0);
}
```

12.5.7　比较 Web 服务器

Web 服务器允许其他机器上的客户得到目录信息、读取文件和运行程序。所有的 Web 服务器都要完成这些基本操作，并且必须遵守 HTTP 协议。

那么服务器之间有什么区别呢？有的服务器容易配置和操作，有的提供了更多的安全特征，有的则快速处理请求或使用较少的内存。其中一个重要的特征是服务器的效率问题。服务器可以同时处理多少个请求？对于每个请求，服务器需要多少系统资源？

本书的 Web 服务器对于每个请求都创建新进程来处理。这是最高效的方法吗？读取文件和目录的请求需要较长的时间，所以服务器没有必要等待这些操作完成，但是有必要用一个新的进程吗？

有第三种方法可以同时运行多个操作。程序可以在一个进程中运行多个任务，这可通过使用线程(thread)来实现。在后面的章节中将学习它的使用。

小　　结

1. 主要内容

- 基于 socket 的客户/服务器程序遵循一个标准框架。服务器接收和处理请求，客户发出请求。
- 服务器建立服务器端 socket。服务器端 socket 有具体的地址，用来接收连接。
- 客户创建和使用客户端 socket。客户并不关心客户端 socket 的地址。
- 服务器可以用两种方法之一处理请求：自己处理请求，或使用 fork 创建新进程来处

理请求。

- Web 服务器是最受欢迎的基于 socket 的程序。Web 服务器处理 3 种类型的请求：返回文件内容、目录列表和运行程序。请求和应答协议称为 HTTP。

2. 下一步做什么

电话呼叫模型不是客户和服务器通信的惟一方式。有些人通过邮件发送定购请求来买商品。使用基于消息的通信系统，每个购物者可以一次处理多个商店的购物，而商店可以同时处理多个顾客的请求。在下一章中，将学习使用明信片模型的网络编程：数据报(Datagram)socket。

3. 习题

12.1　在时间服务的例子中调用了 read 和 write 一次。如果服务器端发送过来的数据分几次到达或超出缓存的大小将会怎样？如果客户端需要多次调用 read，该如何修改？在服务器端，write 的返回值小于字符串的长度，又会怎样？

12.2　修订版的 SIGCHLD 的处理函数是用 waitpid 和一个循环。那么可以在一个循环中使用常规的 wait 来处理多个信号问题吗？

4. 编程练习

12.3　改写本章中的一般服务器模型，使得它被信号中断时，可以重起调用 accept。

12.4　改写 Web 服务器，使得它保留所有请求和返回状态的日志信息。

12.5　当 Web 服务器接收 CGI 程序请求时，服务器将设置一些 CGI 程序的环境变量。找出这些环境变量，并把其中的一些加到 Web 服务器中。Shell 一章中解释了如何设置环境变量。

12.6　Web 服务器可以使用两种方法来识别每个请求所要运行的程序。本章中是以.cgi 作为扩展名来识别要运行的程序。另一种方法是使用路径。特别地，如果请求的路径中包含有目录名 cgi-bin，该程序就被运行。例如，对于/cgi-bin/counter 的请求在该系统下将被服务器执行。改写服务器以支持这种方法。

12.7　改写 Web 服务器使得它可以发送更多的信息。书中例子中的连接给出了典型的头部项的集合。将这些项增加到 Web 服务器中。

12.8　改写 Web 服务器以支持 HEAD 请求。阅读 HTTP 协议获取详细信息。

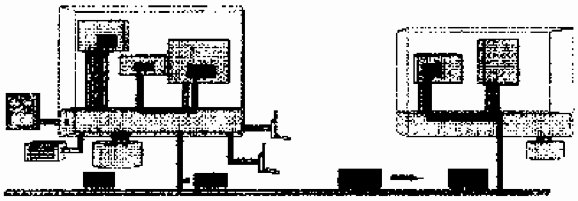
12.9　改写 Web 服务器以支持 POST 请求。阅读 HTTP 协议获取详细信息。

5. 项目

基于本章的内容，可以学习编写下面的 Unix 程序：

httpd、telnetd、fingerd、ftpd

第 13 章　基于数据报(Datagram)的编程：编写许可证服务器[①]



概念与技巧

- 基于数据报的编程，数据报 socket
- TCP 与 UDP
- 许可证服务器
- 软件时间戳(Software ticket)
- 设计健壮系统
- 设计分布式系统
- Unix 域的 socket

相关的系统调用与函数

- socket
- sendto、recvfrom

13.1 软件控制

程序的运行需要内存、CPU 和一些系统资源。操作系统关心这类事情，但是一些程序还需要关心另一件事情：就是程序拥有者的允许。

从合法性的角度讲，需要有一个许可证(license)来保证程序的合法运行，但是有些许可证却是带有限制的。例如，有的许可证是限制同时运行程序的用户数。一个 10 人的许可证可能需要一定的花费，而 50 人的许可证可能需要更多的花费。有些厂商租赁软件许可证，当租赁期到了，程序就不能继续运行。当然除了合法性方面之外，软件也还受到其他一些因素的限制。学校里的计算机房就可能限制每天游戏程序运行的次数。

有些软件拥有者使用诚信机制来限制程序的使用，他们把许可证条款打印在屏幕上或纸上并要求用户遵守协约。其他的则使用特定的技术来实施许可证条款。

---

① 本章的内容基于 Lawrence deLuca 在哈佛职业教育学院担任助教期间编写的讲稿，该讲稿取材于他参与开发的一个产品。

一种实施许可证技术是编写程序来执行许可证控制。通常的做法是设计从一个许可证服务器获得许可的应用程序。该服务器是一个进程，它授权应用程序的运行。

许可证服务器明确许可证条款并且强制执行该条款，如图 13.1 所示。

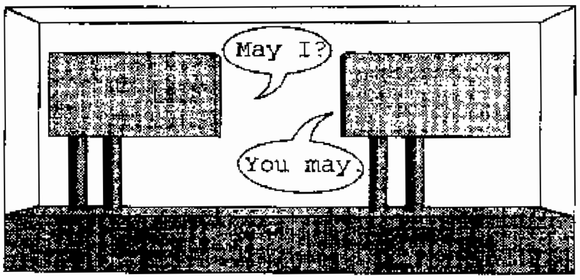


图 13.1　许可证服务器给予许可

请求许可和给予许可需要客户和许可证服务器之间的通信。

许可证服务器是如何工作的？本章将学习一个具体的客户/服务器许可证控制模型。通过该模型的学习，可以接触到另外一种类型的 socket——数据报 socket，另外的一个地址域——Unix 域，一个维持系统状态的网络协议，以及其他一些有关设计安全健壮的客户/服务器系统的技术。

13.2　许可证控制简史

软件控制技术的使用已经有多年的发展历史了。

在单机版的个人计算机时代，对软件的限制是靠特定的磁盘或由隐匿在特定磁道上的密码来实现的。磁盘上的密码很难被复制，并且只有磁盘在驱动器中的时候程序才能运行。如果磁盘丢失了或被损坏了，该程序将不能再运行。人们很快就破解并找到了如何来复制这种特殊磁盘的方法，所以软件厂商发明了硬件密钥。硬件密钥是一个适配器，它可以插在并口、串口或 USB 口上；被许可的程序只有在发现该适配器的时候才能运行。如果硬件密钥丢失，程序将不能运行。

然而网络计算机和多用户系统却引起了新的问题。如果 10 个用户同时想运行计算机或网络上的同一个程序，那么是不是每个用户都需要在服务器的端口上插一个硬件密钥呢？软件厂商们需要一种可靠的并且不需要给合法用户增加额外负担的方法来实施许可证条款。

网络和多用户系统提供了一种新的解决方法：许可证服务器。程序不是从磁盘或密钥取得许可，而是从服务器进程取得。许可证服务器被一台计算机上多个用户共享，服务器进程能够控制程序的使用人数、使用时间、使用地点甚至程序的使用方式。伴随更多的计算机加入因特网，服务器控制对软件和数据访问的需求也更加迫切。

本章中的许可证服务器将实施 n 用户的限制。也就是说，服务器只允许特定数量的程序实例同时运行。

13.3 一个非计算机系统实例：轿车管理系统

一个公司购买了许可证，该许可证限制了同时使用程序的用户数。公司可能拥有比许可证所允许的用户要多的雇员，但是并非所有的雇员要同时使用程序。怎样的一个设计才能满足上面的需求呢？

现实世界中有很多系统，通常由一大群人来共享其中的资源，而这些资源是有限的。这里将分析一个模型：雇员共享公司轿车的问题。一个公司拥有特定数量的轿车，同时还拥有更多数量的雇员。如何控制对轿车的使用呢？

13.3.1 轿车钥匙管理描述

控制轿车的使用是通过控制对轿车钥匙的访问。当想用轿车的时候，必须先得到一把钥匙。如果在钥匙盒中没有钥匙了，将不能使用轿车。如果有可用钥匙，可以先拿一把钥匙，签名，然后使用轿车。使用完毕后，把钥匙放回钥匙盒，在使用轿车名单中划去自己的名字。过程描述如图 13.2 所示。

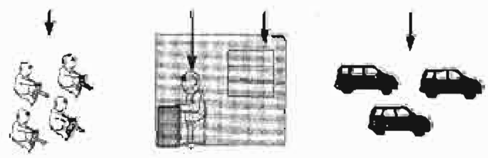


图 13.2 控制对轿车的的使用

签名列表有何作用呢？该系统的目的是通过控制对轿车的使用，使得可用的钥匙一直很充足。人并非是完美的，有时司机会忘记归还钥匙。钥匙管理员可以根据签名列表找到该司机，确定他是否仍在使用车。

轿车使用管理系统是控制软件使用的一个现实模型。在将该系统转换成软件系统之前，需要更为详细地描述该系统。

钥匙管理系统的构成如下：

(1) 钥匙管理中心的地点——到哪儿可以获得钥匙

(2) 钥匙管理员——执行策略的人

(3) 钥匙——需要得到的东西

(4) 签名列表——保存钥匙和找回钥匙的记录

13.3.2 用客户/服务器方式管理轿车

在给出轿车钥匙管理系统的构成后，下面将用客户/服务器语言来描述该系统。

(1) 服务器和客户

谁是服务器和谁是客户？钥匙管理员拥有司机需要的钥匙。用网络术语来描述，钥匙

管理员是服务器，司机是客户。

(2) 协议

所用的协议是什么？交互的事务是什么？轿车钥匙管理协议包括两个主要的事务。

- 获得钥匙

客户：你好，我需要一把钥匙。

服务器：这里只有 5 号钥匙，没有别的钥匙了。

- 归还钥匙

客户：我已经用好 5 号钥匙。

服务器：谢谢。

(3) 通信系统

客户和服务器如何通信？在该系统中，人们通过对话来传递简单信息。

(4) 数据结构

司机和钥匙管理员需要什么样的数据结构呢？钥匙管理员保留一个用户签名列表，一把钥匙对应列表的一项。当一个用户取走一把钥匙，钥匙管理员在该项记下用户的名字，当用户归还钥匙时，管理员把司机的名字从列表中擦去。下表解释了用户签名列表：

签名列表

钥匙 #	司　机
1	adam@sales
2	
3	carol@support
4	

如果在签名列表中没有司机与钥匙号对应，表示该钥匙是可用的。反之，表示不可用。

13.4　许可证管理

本节将钥匙管理系统的思想运用到许可证管理系统中。

13.4.1　许可证服务系统：它做些什么

图 13.3 描述了人们试图运行许可证程序的过程。工作如下：

(1) 用户 U 运行被许可的程序 P；

(2) 程序 P 向服务器 S 请求运行许可；

(3) 服务器检查当前运行程序 P 的用户数；

(4) 如果上限未达到，S 给予许可，程序 P 运行；

(5) 如果达到上限，S 拒绝许可，程序 P 告诉 U 稍后再试。

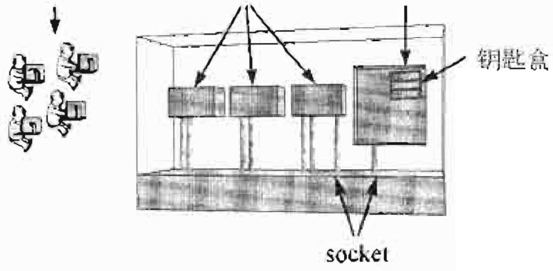


图 13.3 控制软件的使用

许可证服务系统与轿车钥匙服务系统稍有不同。在轿车系统中，司机向钥匙管理员请求许可；而在这里，程序向服务器请求许可。这就像司机请求轿车，而轿车向钥匙管理员请求钥匙[①]。

程序的创建者要编写所有的程序：应用程序和服务器。这两个程序作为一个系统。服务器给予应用程序运行许可和实施许可证条款。如果许可证服务器不在运行，应用程序将得不到许可，不能运行。

这里以轿车钥匙服务系统作为模型进行了讨论。那么如何将这种思想运用到软件系统中呢？被许可程序如何从服务器得到许可？服务器如何给予许可？软件系统和轿车钥匙系统的等价之处在哪里呢？

13.4.2 许可证服务系统：如何工作

(1) 票据模型

钥匙管理员负责分发钥匙，许可证服务器发布什么呢？以电影院和棒球场为例，付钱获得进场许可，然后得到一张入场票据。与此类似，这里的许可证服务器发布的是数字票据。这种票据又是什么样子的呢？客户和服务器交换字符串，所以票据是字符串，其格式如下：

```
pid.ticketnumber 例如：6589.3
```

每张票据包括持有该票据进程的 PID 以及票据编号。在票据中包含 PID 的原因其实就和在飞机票上印上名字的道理是一样的。票据中的 PID 标识票据的使用者，在票据丢失时，可以帮助找回票据。进程可能丢失票据吗？

(2) 服务器和客户

谁是客户和谁是服务器？许可证服务器持有程序需要的资源：票据。用网络术语，许可证服务器是服务器，而应用程序是客户。

(3) 协议

协议是什么？交互的事务是什么？这里的票据管理协议包括下面的两个主要事务：

---

① 这个概念并不难理解。随着设备越来越先进，连接越来越方便，很可能在不久的将来就可以通过你的收音机询问服务器是否有播放你最喜欢歌曲的许可。

- 获取钥匙

客户：HELO mypid

服务器：TICK ticketid or FAIL no tickets

- 归还钥匙

客户：GBYE ticketid

服务器：THNX message

这里定义了基于文本的协议，协议中使用了 4 个字符的简单命令，这与前面的 Web 服务器所用的命令类似。

(4) 通信系统

上述简短的文本信息如何在客户和服务器之间传送？在本文后面将讨论该问题。

(5) 数据结构

客户和服务器之间所用的数据结构是什么？这里使用整型数组作为签名列表。数组的每个入口对应一张票据。当一个客户取走了一张票，管理员将客户对应的 PID 写到该入口。如下表：

签名列表	
tick #	**process**
1	1234
2	0
3	6589
4	0

如果数组中的某个元素值是 0，表明该票据可用。否则，表明该票据正在被使用。

13.4.3　一个通信系统的例子

客户如何请求票据？服务器如何发布票据？这涉及到进程间的通信形式。客户和服务器之间通过短消息通信。服务器必须接收，处理和应答来自多个客户的请求。目前，哪些技术是可以使用的呢？本文前面学习的信号和管道机制可以考虑，但是信号太短了，而管道只连接相关联的进程。所以，使用 socket 是最明显的答案。而对于 socket，也有两种不同的选择。一种是基于流的 socket，它是用来连接不相关的进程的。另一种 socket 被称为数据报 socket，或称为 UDP，对于现在的项目，这种 socket 是更好的选择。

13.5　数据报 socket

流 socket 传送数据就跟电话网中传送声音一样，客户先建立连接，然后使用该连接进行单向、双向或类似管道的字节流传送。

数据报通信则与从一个邮箱到另一个邮箱发送包裹类似。客户不必建立连接，只要向特定的地址发送消息，而服务器进程在该地址接收消息。

流 socket 使用的网络协议叫 TCP 即传输控制协议（Transmission Control Protocol）。数据报 socket 叫 UDP 即用户数据报协议（User Datagram Protocol）。它们的区别是什么呢？何时选择何种 socket 是较好的呢？在程序中如何使用数据报 socket 呢？

13.5.1　流与数据报的比较

socket 是如何工作的？数据如何在 Internet 上传输的？当用户向流 socket 写数据时，内核要做些什么？这与向数据报 socket 写数据有何区别？

从一个流 socket 传输到另一个流 socket 的数据流，看上去是连续的、无缝的，实际上这是一种错觉。Internet 连接要把数据分割成独立的数据包。网络数据传输过程如图 13.4 所示。

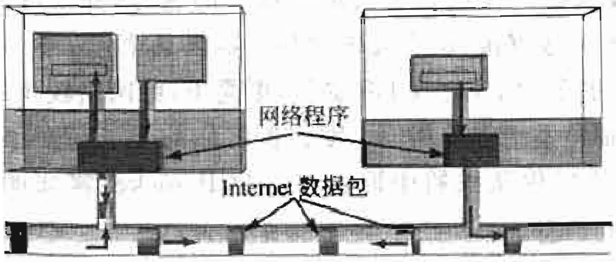


图 13.4　Internet 包传载数据

在现实世界中，有很多将一大块数据分割成若干较小的数据块的例子。假设要通过快递传送 100 页的文档。当快递公司要求你使用只能装 20 页的信封时，该怎么办？可以把文档分割成 5 个包裹，每个单独地打包和附上地址。然后把这 5 个独立的包裹放到邮箱中，运输系统将负责把它们送到目的地。在接收端，接受者打开这 5 个包裹，并把它们按顺序重组成原来的文档。

Internet 就像上述运输系统一样，它上面传输的数据必须符合大小的限制。大块的数据被分割成小块的数据来传输，接收端必须以正确的顺序重组数据块。但通信过程可以被连接和中断，如图 13.5 所示。

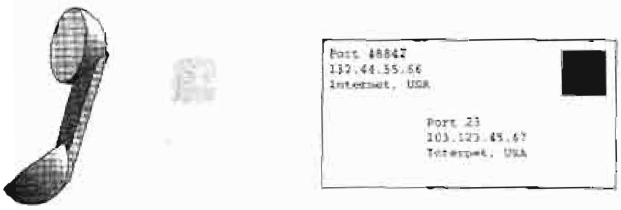


图 13.5　通信可以被连接和中断

流 socket 负责分割、排序、重组的所有工作。数据报 socket 则不会。下表给出了它们之间的区别。

TCP	UDP
流	**数据报**
分片/重组	否
排序	否
可靠的	可能未到达
连接的	多个发送者

在流 socket 中，内核将大的数据块分割成带编号的数据片。位于接收机器上的内核按顺序接收数据片，重组成发送者所发送的原始数据。在数据报 socket 中，内核并不给数据加编号标签，在目的地也不重组。

流 socket 对传送负责，数据报 socket 则不。流 socket 的接收端检查数据片的顺序来确保数据完整到达。接收端提示发送端丢失的数据片的编号，并等待丢失数据片的重传。数据报 socket 并不检查丢失数据报，也不要求重传。如果某个包丢失在 Internet 中，它只是不到达。TCP 比 UDP 做更多的工作。UDP 更快、更简单，给网络较少的负荷。

UDP 接收消息跟邮箱系统方式相同：发送者将你的地址写在信件上，邮件系统负责将信件送到你的邮箱中，然后你从邮箱中取出信。TCP socket 需要明确调用 accept、read 和 close 来读取远端进程的消息。

UDP 正好适合这里的应用。客户发送短消息来获得许可，服务器发送短消息给予或拒绝许可。客户和服务器的交互不需要建立连接，不需要分片和重组。可靠性甚至是不要的。如果请求或票据丢失，客户可以再次请求。在任何事件中，服务器和客户就像在一台机器上或者一个网络的同一部分，所以丢失数据报的风险较小。

UDP 对于 Web 服务器和 e-mail 服务是一个较差的选择。Web 服务器和 e-mail 信息可能是大的文件，这些字节流必须完全和按序到达目的地。UDP 对于允许丢失帧的声音和视频流是较好的选择。

13.5.2 数据报编程

数据报与邮件网络系统类似，包括 3 个主要部分：目的地址、返回地址和消息，如图 13.6 所示。

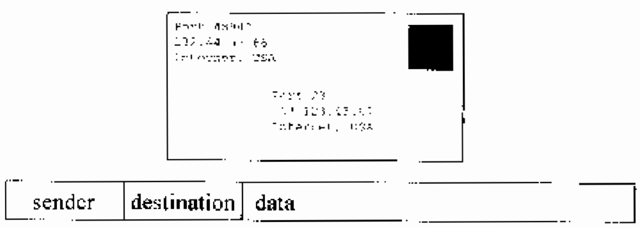


图 13.6 数据报的 3 部分

数据报 socket 可以被理解成一个内部有数据的盒子，发送者给出目的 socket 的地址。网络将数据从发送者发送到目的 socket。接收进程从该 socket 中读取数据。程序使用 sendto 发送数据和 recvfrom 来读取数据，如图 13.7 所示。

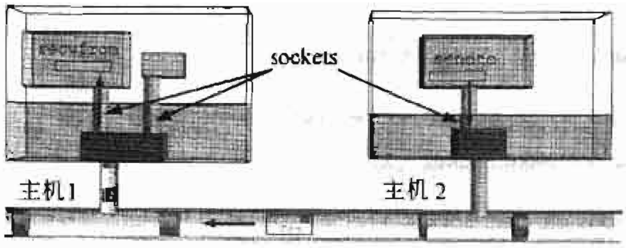


图 13.7 使用 sendto 和 recvfrom

1. 接收数据报

程序 dgrecv.c 是一个简单的基于数据报的服务器。dgrecv.c 使用命令行传过来的端口号建立 socket，然后进入循环，接收和打印从客户端发来的数据报：

```
/****************************************************************
 * dgrecv.c - datagram receiver
 *            usage: dgrecv portnum
 *            action: listens at the specfied port and reports messages
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define oops(m,x) { perror(m);exit(x);}

int make_dgram_server_socket(int);
int get_internet_address(char * , int, int * , struct sockaddr_in * );
void say_who_called(struct sockaddr_in * );

int main(int ac, char * av[])
{
    int         port;                   /* use this port */
    int         sock;                   /* for this socket */
    char        buf[BUFSIZ];            /* to receive data here */
    size_t      msglen;                 /* store its length here */
    struct      sockaddr_in saddr;      /* put sender's address here */
    socklen_t saddrlen;                 /* and its length here */

    if ( ac == 1 || (port = atoi(av[1])) <= 0 ){
      fprintf(stderr,"usage: dgrecv portnumber\n");
      exit(1);
```

```
    }

  /* get a socket and assign it a port number */

    if( (sock = make_dgram_server_socket(port)) == -1 )
        oops("cannot make socket",2);

  /* receive messaages on that socket */

    saddrlen = sizeof(saddr);
    while( (msglen = recvfrom(sock,buf,BUFSIZ,0,&saddr,&saddrlen))>0 )
    {
        buf[msglen] = '\0';
        printf("dgrecv: got a message: %s\n", buf);
        say_who_called(&saddr);
    }
    return 0;
}
void say_who_called(struct sockaddr_in *addrp)
{
    char    host[BUFSIZ];
    int     port;

    get_internet_address(host,BUFSIZ,&port,addrp);
    printf(" from: %s: %d\n", host, port);
}
```

辅助函数 make_dgram_server_socket 和 get_internet_address 将在后面给出的文件 dgram.c 中定义。在数据报 socket 中接收消息比从流 socket 接收简单。recvfrom 函数阻塞直到数据报到达。当数据报到达时，消息内容、返回地址和其长度将被复制到缓存中。

2. 发送数据报

程序 dgsend.c 发送数据报。dgsend.c 创建一个 socket，然后用它发送消息到以命令行参数传入的特定的主机和端口号。

```
/******************************************************************
 * dgsend.c  -  datagram sender
 *               usage: dgsend hostname portnum "message"
 *              action: sends message to hostname:portnum
 */

#include   <stdio.h>
#include   <stdlib.h>
#include   <sys/types.h>
#include   <sys/socket.h>
#include   <netinet/in.h>

#define oops(m,x) { perror(m);exit(x);}
```

```
int make_dgram_client_socket();
int make_internet_address(char * ,int, struct sockaddr_in * );

int main(int ac, char * av[])
{
    int       sock;                   /* use this socket to send */
    char      * msg;                  /* send this messag */
    struct    sockaddr_in   saddr;    /* put sender's address here */

    if ( ac != 4 ){
        fprintf(stderr,"usage: dgsend host port 'message'\n");
        exit(1);
    }
    msg = av[3];

  /* get a datagram socket */

    if( (sock = make_dgram_client_socket()) == -1 )
        oops("cannot make socket",2);

  /* combine hostname and portnumber of destination into an address */

    make_internet_address(av[1], atoi(av[2]), &saddr)

  /* send a string through the socket to that address */

    if ( sendto(sock, msg, strlen(msg), 0, &saddr,sizeof(saddr)) == -1)
        oops("sendto failed", 3);
    return 0;
}
```

sendto 函数将缓存中的内容发送到特定地址的 socket。

3. 辅助函数

创建 socket 和 socket 地址的细节被封装在 dgram.c 中：

```
/**********************************************************************
 * dgram.c
 * support functions for datagram based programs
 */

#include  <stdio.h>
#include  <unistd.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <arpa/inet.h>
#include  <netdb.h>
#include  <string.h>

#define  HOSTLEN  256
```

```
int make_internet_address();

int make_dgram_server_socket(int portnum)
{
    struct   sockaddr_in saddr;    /* build our address here */
    char     hostname[HOSTLEN];    /* address */
    int      sock_id;              /* the socket */

    sock_id = socket(PF_INET, SOCK_DGRAM, 0);  /* get a socket */
    if ( sock_id ==  -1 ) return -1;

    /** build address and bind it to socket ** /

    gethostname(hostname, HOSTLEN);            /* where am I ? */
    make_internet_address(hostname, portnum, &saddr);

    if( bind(sock_id,(struct sockaddr * )&saddr, sizeof(saddr)) != 0 )
        return -1;

    return sock_id;
}
int make_dgram_client_socket()
{
    return socket(PF_INET, SOCK_DGRAM, 0);
}

int make_internet_address(char * hostname, int port, struct sockaddr_in * addrp)
/*
 * constructor for an Internet socket address, uses hostname and port
 * (host,port)  ->  * addrp
 */
{
    struct hostent * hp;

    bzero((void * )addrp, sizeof(struct sockaddr_in));
    hp = gethostbyname(hostname);
    if ( hp == NULL ) return -1;
    bcopy((void * )hp->h_addr, (void * )&addrp->sin_addr, hp->h_length);
    addrp->sin_port = htons(port);
    addrp->sin_family = AF_INET;
    return 0;
}

int get_internet_address(char * host, int len, int * portp, struct sockaddr_in * addrp)
/*
 * extracts host and port from an internet socket address
 ** addrp -> (host,port)
 */
{
    strncpy(host, inet_ntoa(addrp->sin_addr), len );
```

```
    *portp = ntohs(addrp->sin_port);
    return 0;
}
```

创建一个数据报 socket 与创建一个流 socket 类似。其不同点在于，这里设置 socket 的类型为 SOCK_DGRAM，而且不要调用 listen 函数。

4. 编译和测试

```
$ cc dgrecv.c dgram.c -o dgrecv
$ ./dgrecv 4444 &
[1] 19383
$ cc dgsend.c dgram.c -o desend
$ ./dgsend host2 4444 "testing 123"
dgrecv:got a message: testing 123
  from:10.200.75.200:1041
$ ps
PID     TTY     TIME        CMD
14599   pts/3   00:00:00    bash
19383   pts/3   00:00:00    dgrecv
19393   pts/3   00:00:00    ps
$
```

编译服务器，并启动它，使得它监听在端口 4444。然后编译和运行客户，使客户发送字符串到端口 4444。服务器接收消息，打印消息，并且打印消息的返回地址。客户 socket 拥有主机地址和端口号，内核随机地给它分配了一个端口号 1041。ps 进程显示了服务器正在运行。

13.5.3　sendto 和 recvfrom 的小结

	sendto
目标	从 socket 发送消息
头文件	#include <sys/types.h> #include <sys/socket.h>
函数原型	nchars = sendto(int socket, const void *msg, size_t len, int flags, const struct sockaddr *dest, socklen_t dest_len);
参数	socket　　socket id msg　　发送的字符类型的数组 len　　发送的字符数 flags　　比特的集合，设置发送属性，0 表示普通 dest　　指向远端 socket 地址的指针 dest_len　　地址长度
返回值	−1　　遇到错误 nchars　　发送的字符数

sendto 从源 socket 发送数据报到目的 socket。前 3 个参数与 write 的参数类似：要发送的 socket、保存要发送字符串的数组以及发送的字符数。与 write 类似，sendto 返回实际发送的字符数。flags 参数表明发送的各种属性；具体可参见 Unix 版本的帮助信息。最后两个参数给出发送目的地的 socket 地址。如果是 Internet 类型的地址，socket 地址包含目的主机的 IP 地址和端口号，而其他类型的地址则包含其他的成员。

recvfrom		
目标	从 socket 接收消息	
头文件	#include <sys/types.h> #include <sys/socket.h>	
函数原型	nchars = recvfrom(int socket, const void * msg, size_t len, int flags, const struct sockaddr * sender, socklen_t * sender_len);	
参数	socket	socket id
	msg	字符类型的数组
	len	接收的字符数
	flags	表示接收属性的比特的集合，0 表示普通
	sender	指向远端 socket 的地址的指针
	sender_len	地址长度
返回值	−1	遇到错误
	nchars	接收的字符数

recvfrom 从 socket 读取数据报。前 3 个参数与 read 类似：所要读取的 socket、存放字符的数组以及要读取的字符数。与 read 类似，recvfrom 返回实际接收的字符数。flags 指出了接收时所用的各种属性；具体可参见 Unix 系统的帮助信息。通过最后两个参数，可以获得发送者的地址。发送 socket 的地址将被存放在由第一个参数指向的结构中，地址长度存放在由第二个参数指向的整型值中。地址的长度必须提供给 recvfrom 函数；如果实际的地址是不同的长度，它将更改该值。如果第一个参数指针为空值，发送者地址将不被记录。

13.5.4　数据报应答

程序 dgsend.c 和 dgrecv.c 显示了如何从客户发送数据给服务器。服务器如何给客户发送应答呢？在现实世界中，假设有人给你发送了一封晚宴的邀请函，你将如何给出答复呢？这很简单：你只要给邀请函上的返回地址回信就行了。

程序 dgrecv2.c 从客户处接收消息并且发送谢谢作为应答：

```
/****************************************************************
 * dgrecv2.c  - datagram receiver
 *              usage: dgrecv portnum
 *             action: receives messages, prints them, sends reply
 */

#include  <stdio.h>
#include  <stdlib.h>
```

```
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define oops(m,x) { perror(m);exit(x);}

int make_dgram_server_socket(int);
int get_internet_address(char* ,int, int* struct sockaddr_in* );
void say_who_called(struct sockaddr_in * );
void reply_to_sender(int,char * ,struct sockaddr_in * , socklen_t);

int main(int ac, char * av[])
{
    int         port;                   /* use this port */
    int         sock;                   /* for this socket */
    char        buf[BUFSIZ];            /* to receive data here */
    size_t      msglen;                 /* store its length here */
    struct      sockaddr_in saddr;      /* put sender's address here */
    socklen_t   saddrlen;               /* and its length here */

    if ( ac == 1 || (port = atoi(av[1])) <= 0 ){
        fprintf(stderr,"usage: dgrecv portnumber\n");
        exit(1);
    }

  /* get a socket and assign it a port number */

    if( (sock = make_dgram_server_socket(port)) == -1 )
        oops("cannot make socket",2);

  /* receive messaages on that socket */

    saddrlen = sizeof(saddr);
    while( (msglen = recvfrom(sock,buf,BUFSIZ,0, &saddr,&saddrlen))>0 ) {
       buf[msglen] = '\0';
       printf("dgrecv: got a message: %s\n", buf);
       say_who_called(&saddr);
       reply_to_sender(sock,buf,&saddr,saddrlen);
    }
    return 0;
}
void reply_to_sender(int sock,char * msg,struct sockaddr_in * addrp,socklen_t len)
{
    char reply[BUFSIZ + BUFSIZ];

    sprintf(reply, "Thanks for your %d char message\n", strlen(msg));
```

```
    sendto(sock, reply, strlen(reply), 0, addrp, len);
}
void say_who_called(struct sockaddr_in * addrp)
{
    char    host[BUFSIZ];
    int     port;

    get_internet_address(host,BUFSIZ,&port,addrp);
    printf(" from: %s:%d\n", host, port);
}
```

发送者程序当然也要改变以接收应答。这作为练习由读者来完成。

13.5.5 数据报小结

数据报是从一个 socket 发送到另一个的短消息。发送者使用 sendto 来指定消息、长度和目的地。接收者使用 recvfrom 接收消息。数据报和带有地址的 Internet 上传输的数据包的基本结构接近。因此，数据报给内核网络功能和网络流量增加的负荷较少。由于数据报可能在传输中丢失，也有可能不按顺序地到达，所以它通常用对简单和高效的要求比完整性和一致性更为重要的应用中。

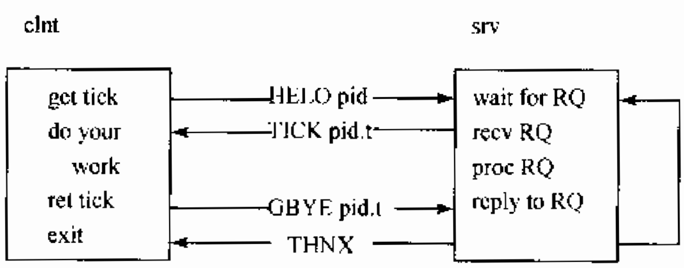
本章的许可证服务器是一种简单的消息传输，它用一个服务器来接收、处理短消息和发送请求，所以数据报是一个合适的选择。

13.6 许可证服务器版本 1.0

下面，回到许可证服务器项目上来。这里的服务器限制了程序同时运行的实例数目。当用户要运行受限制的程序时，该进程要向服务器请求运行许可。

如果并没有很多人正在用该程序，服务器发送票据给进程，给予许可证。如果达到了最大的程序实例数，服务器发送无可用票据的消息给客户，并且通知客户稍后再试或者购买支持更多用户版本的软件。被许可程序和服务器之间通过数据报来通信。

客户和服务器的运行流程及其交互过程如下。



客户和服务器分别由两个文件组成：短的文件包含 main 函数，长的文件包含票据管理函数。下面将分析客户和服务器程序。

13.6.1　客户端版本 1

```
/*******************************************************************
 * lclnt1.c
 * License server client version 1
 * link with lclnt_funcs1.o dgram.o
 */

#include <stdio.h>

int main(int ac, char *av[])
{
    setup();
    if (get_ticket() != 0 )
        exit(0);

    do_regular_work();

    release_ticket();
    shut_down();

}
/*******************************************************************
 * do_regular_work the main work of the application goes here
 */
do_regular_work()
{
    printf("SuperSleep version 1.0 Running - Licensed Software\n");
    sleep(10); /* our patented sleep algorithm */
}
```

客户端的最上层代码遵照了上一节中所小结的原则。客户得到票据，进行处理，释放票据，然后退出。该许可证例程是 Unix 中 sleep 程序的特殊版本，若不满意标准 sleep 版本的工作，也可以购买许可证来使用此版本的程序。当然还要运行许可证服务器，否则该 sleep 程序将拒绝运行。其中辅助函数在 lclnt_funcs1.c 中：

```
/*******************************************************************
 * lclnt_funcs1.c: functions for the client of the license server
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

```
/*
 * Important variables used throughout
 */
static int pid = -1;                    /* Our PID */
static int sd = -1;                     /* Our communications socket */
static struct sockaddr serv_addr;       /* Server address */
static socklen_t serv_alen;             /* length of address */
static char ticket_buf[128];            /* Buffer to hold our ticket */
static have_ticket = 0;                 /* Set when we have a ticket */

#define MSGLEN          128             /* Size of our datagrams */
#define SERVER_PORTNUM 2020             /* Our server's port number */
#define HOSTLEN         512
#define oops(p) { perror(p); exit(1) ; }

char *do_transaction();

/*
 * setup: get pid, socket, and address of license server
 * IN no args
 * RET nothing, dies on error
 * notes: assumes server is on same host as client
 */
setup()
{
    char    hostname[BUFSIZ];

    pid = getpid();                         /* for ticks and msgs */
    sd = make_dgram_client_socket();        /* to talk to server */
    if ( sd == -1 )
        oops("Cannot create socket");
    gethostname(hostname, HOSTLEN);         /* server on same host */
    make_internet_address(hostname, SERVER_PORTNUM, &serv_addr);
    serv_alen = sizeof(serv_addr);
}

shut_down()
{
    close(sd);
}
/***********************************************************************
 * get_ticket
 * get a ticket from the license server
 * Results: 0 for success, -1 for failure
 */
int get_ticket()
```

```
{
    char * response;
    char buf[MSGLEN];

    if(have_ticket)                              /* don't be greedy */
        return(0);

    sprintf(buf, "HELO %d", pid);                /* compose request */

    if ( (response = do_transaction(buf)) == NULL )
        return( -1);

    /* parse the response and see if we got a ticket.
     * on success, the message is: TICK ticket-string
     * on failure, the message is: FAIL failure-msg
     */
    if ( strncmp(response, "TICK", 4) == 0 ){
        strcpy(ticket_buf, response + 5);        /* grab ticket-id */
        have_ticket = 1;                         /* set this flag */
        narrate("got ticket", ticket_buf);
        return(0);
    }

    if ( strncmp(response,"FAIL",4) == 0)
        narrate("Could not get ticket",response);
    else
        narrate("Unknown message:", response);

    return( -1);
} /* get_ticket */

/**********************************************************************
 * release_ticket
 * Give a ticket back to the server
 * Results: 0 for success, -1 for failure
 */
int release_ticket()
{
    char buf[MSGLEN];
    char * response;

    if(! have_ticket)                              /* don't have a ticket */
        return(0);                                 /* nothing to release */

    sprintf(buf, "GBYE %s", ticket_buf);           /* compose message */
    if ( (response = do_transaction(buf)) == NULL )
        return( -1);

    /* examine response
```

```
     * success: THNX info - string
     * failure: FAIL error - string
     */
    if ( strncmp(response, "THNX", 4) == 0 ){
        narrate("released ticket OK","");
        return 0;
    }
    if ( strncmp(response, "FAIL", 4) == 0)
        narrate("release failed", response + 5);
    else
        narrate("Unknown message:", response);
    return( - 1);
} /* release_ticket */

/****************************************************************
 * do_transaction
 * Send a request to the server and get a response back
 * IN msg_p              message to send
 * Results: pointer to message string, or NULL for error
 *                       NOTE: pointer returned is to static storage
 *                       overwritten by each successive call.
 * note: for extra security, compare retaddr to serv_addr (why?)
 */
char * do_transaction(char * msg)
{
    static char buf[MSGLEN];
    struct sockaddr retaddr;
    socklen_t    addrlen = sizeof(retaddr);
    int ret;


    ret = sendto(sd, msg, strlen(msg), 0, &serv_addr, serv_alen);
    if ( ret == - 1 ){
        syserr("sendto");
        return(NULL);
    }
    /* Get the response back */
    ret = recvfrom(sd, buf, MSGLEN, 0, &retaddr, &addrlen);
    if ( ret == - 1 ){
        syserr("recvfrom");
        return(NULL);
    }

    /* Now return the message itself */
    return(buf);
```

```
} /* do_transaction */

/*************************************************************************
 * narrate: print messages to stderr for debugging and demo purposes
 * IN msg1, msg2 : strings to print along with pid and title
 * RET nothing, dies on error
 */
narrate(char * msg1, char * msg2)
{
    fprintf(stderr,"CLIENT [%d]: %s %s\n", pid, msg1, msg2);
}
syserr(char * msg1)
{
    char    buf[MSGLEN];
    sprintf(buf,"CLIENT [%d]: %s", pid, msg1);
    perror(buf);
}
```

get_ticket 和 release_ticket 是程序中的主要函数。它们都遵循下面的规则：产生短请求、发送消息给服务器、等待服务器的应答，然后检查应答和根据应答采取行动。

get_ticket 通过发送命令 HELLO 以及紧跟其后的 PID 来请求票据。服务器通过发送 TICK ticket-id 接收请求。服务器发送 FAIL explanation 拒绝请求。

release-ticket 通过发送命令 GBYE ticket-id 返回票据。如果票据是合法的，服务器将发送 THNX greeting 消息作为应答。如果票据不合法，服务器发送 FAIL explanation 消息。

为何票据可能是不合法的？本文的后面将讨论该问题。

13.6.2 服务器端版本 1

```
/*************************************************************************
 * lserv1.c
 * License server server program version 1
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <sys/errno.h>
#define MSGLEN          128             /* Size of our datagrams */

int main(int ac, char * av[])
{
    struct sockaddr_in client_addr;
    socklen_t addrlen = sizeof(client_addr);
```

```
    char        buf[MSGLEN];
    int         ret;
    int         sock;

    sock = setup();

    while(1) {
       addrlen = sizeof(client_addr);
       ret = recvfrom(sock,buf,MSGLEN,0,&client_addr,&addrlen);
       if ( ret != -1 ){
            buf[ret] = '\0';
            narrate("GOT:", buf, &client_addr);
            handle_request(buf,&client_addr,addrlen);
       }
       else if ( errno != EINTR )
            perror("recvfrom");
    }
}
```

许可证服务器的主函数是一个循环，主要包括接收客户请求，处理请求和发送应答。其中处理请求的代码包含在 lserv_funcs1.c 文件中。

```
/****************************************************************
 * lsrv_funcs1.c
 * functions for the license server
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <netdb.h>
#include    <signal.h>
#include    <sys/errno.h>

#define SERVER_PORTNUM 2020              /* Our server's port number */
#define MSGLEN          128              /* Size of our datagrams */
#define TICKET_AVAIL    0                /* Slot is available for use */
#define MAXUSERS        3                /* Only 3 users for us */
#define oops(x) { perror(x); exit(-1); }

/****************************************************************
 * Important variables
 */
int ticket_array[MAXUSERS];     /* Our ticket array */
int sd = -1;                    /* Our socket */
int num_tickets_out = 0;        /* Number of tickets outstanding */
```

```
char * do_hello();
char * do_goodbye();

/*****************************************************************
 * setup() - initialize license server
 */
setup()
{
    sd = make_dgram_server_socket(SERVER_PORTNUM);
    if ( sd == -1 )
        oops("make socket");
    free_all_tickets();
    return sd;
}
free_all_tickets()
{
    int        i;

    for(i = 0; i<MAXUSERS; i++)
        ticket_array[i] = TICKET_AVAIL;
}

/*****************************************************************
 * shut_down() - close down license server
 */
shut_down()
{
    close(sd);
}

/*****************************************************************
 * handle_request(request, clientaddr, addrlen)
 * branch on code in request
 */
handle_request(char *req,struct sockaddr_in *client, socklen_t addlen)
{
    char     *response;
    int      ret;

    /* act and compose a response */
    if ( strncmp(req, "HELO", 4) == 0 )
        response = do_hello(req);
    else if ( strncmp(req, "GBYE", 4) == 0 )
        response = do_goodbye(req);
    else
        response = "FAIL invalid request";
```

```
    /* send the response to the client */
    narrate("SAID:", response, client);
    ret = sendto(sd, response, strlen(response),0,client, addlen);
    if ( ret ==  -1 )
        perror("SERVER sendto failed");
}

/***********************************************************************
 * do_hello
 * Give out a ticket if any are available
 * IN msg_p              message received from client
 * Results: ptr to response
 * Note: returnis in static buffer verwritten by each call
 * NOTE: return is in static buffer overwritten by each call
 */
char *do_hello(char *msg_p)
{
    int x;
    static char replybuf[MSGLEN];

    if(num_tickets_out >= MAXUSERS)
        return("FAIL no tickets available");
    /* else find a free ticket and give it to client */

    for(x = 0; x<MAXUSERS && ticket_array[x] != TICKET_AVAIL; x++) ;

    /* A sanity check - should never happen */
    if(x == MAXUSERS) {
        narrate("database corrupt","",NULL);
        return("FAIL database corrupt");
    }

    /* Found a free ticket. Record "name" of user (pid) in array.
     * generate ticket of form: pid.slot
     */
    ticket_array[x] = atoi(msg_p + 5); /* get pid in msg */
    sprintf(replybuf, "TICK %d.%d", ticket_array[x], x);
    num_tickets_out++;
    return(replybuf);
} /* do_hello */

/***********************************************************************
 * do_goodbye
 * Take back ticket client is returning
 * IN msg_p              message received from client
 * Results: ptr to response
```

```
 * Note: return is in static buffer over written by each call
 */
char * do_goodbye(char * msg_p)
{
    int pid, slot;          /* components of ticket */

    /* The user's giving us back a ticket. First we need to get
     * the ticket out of the message, which looks like:
     *
     * GBYE pid.slot
     */
    if((sscanf((msg_p + 5), "%d.%d", &pid, &slot) != 2) ||
      (ticket_array[slot] != pid)) {
        narrate("Bogus ticket", msg_p+5, NULL);
        return("FAIL invalid ticket");
    }

    /* The ticket is valid. Release it. */
    ticket_array[slot] = TICKET_AVAIL;
    num_tickets_out--;

    /* Return response */
    return("THNX See ya!");
} /* do_goodbye */

/************************************************************************
 * narrate() - chatty news for debugging and logging purposes
 */
narrate(char * msg1, char * msg2, struct sockaddr_in * clientp)
{
    fprintf(stderr,"\t\tSERVER: %s %s ", msg1, msg2);
    if ( clientp )

    fprintf(stderr,"(%s:%d)",inet_ntoa(clientp->sin_addr),
                ntohs(clientp->sin_port) );
    putc('\n', stderr);
}
```

3 个重要的函数解释如下。

(1) handle_request

请求由 4 个字符的命令带一个参数构成。服务器先检查命令，然后调用对应的函数。即使命令不合法，服务器也必须发送应答，否则客户会一直阻塞下去。

(2) do_hello

HELO 命令用来请求票据。服务器查找票据数组寻找空闲的项。如果某项的 PID 值为 0，表示这是一个可用票据。服务器使用独立的变量 num_tickets_out 来节省时间。服务器

接收请求后，可以通过查找表来寻找空闲项，而该变量可以指明何时表已经满了，这样就不必做查找了。

(3) do_goodbye

GBYE 命令是返回票据的请求。票据是一个由 PID 和票据编号构成的字符串。服务器将票据的 PID 和票据编号与签出列表(sign-out list)中的值进行比较，如果数据一致，服务器从签出列表中清除该项的值并且致谢客户。如果不一致，则一定是什么地方发生了错误。

如果你是飞机场的检票员，如果某个客户给出存根的编号和名字在数据库中不存在，你就可能会问："你是从哪里得到这张票的，你是谁?"后面将讨论伪造票据问题。下面来测试这个版本的程序。

13.6.3 测试版本 1

编译服务器端程序并在后台运行它。

```
$ cc lserv1.c lserv_funcs1.c dgram.c - o lserv1
$ ./lserv1&
[1] 25738
```

编译好客户端程序，然后同时运行 4 个实例：

```
$ cc lclnt1.c lclnt_funcs1.c dgram.c - o lclnt1
$ ./lclnt1 & ./lclnt1 & ./lclnt1 & ./lclnt1 &

          SERVER:GOT:HELO 25912(10.200.75.200:1053)
          SERVER:SAID:TICK 25912.0(10.200.75.200:1053)
CLIENT[25912]:got ticket 25912.0
SuperSleep version 1.0 Running   Licensed Software
          SERVER:GOT:HELO 25913(10.200.75.200:1054)
          SERVER:SAID:TICK 25913.1(10.200.75.200:1054)
CLIENT[25913]:got ticket 25913.1
SuperSleep version 1.0 Running - Licensed Software
          SERVER:GOT:HELO 25915(10.200.75.200:1055)
          SERVER:SAID:TICK 25915.2(10.200.75.200:1055)
CLIENT[25915]:got ticket 25915.2
SuperSleep version 1.0 Running - Licensed Software
          SERVER:GOT:HELO 25914(10.200.75.200:1059)
          SERVER:SAID:FAIL no tickets available (10.200.75.200:1059)
CLIENT[25914]:Could not get ticket FAIL no tickets available (10.200.75.200:1059)
          SERVER:GOT:GBYE 25912.0(10.200.75.200:1053)
          SERVER: SAID:THNX See ya! (10.200.75.200:1053)
CLIENT[25912]:released ticket OK
          SERVER:GOT:GBYE 25913.1(10.200.75.200:1054)
          SERVER: SAID:THNX See ya! (10.200.75.200:1054)
CLIENT[25913]:released ticket OK
```

```
        SERVER:GOT:GBYE 25915.2(10.200.75.200:1055)
        SERVER: SAID:THNX See ya! (10.200.75.200:1055)
CLIENT[25915]:released ticket OK
```

实际运行的程序可能有非常不同的结果。尽管如此，从中可以看到服务器如何接收请求、给出票据以及客户端如何获取票据并开始工作的：可以看到进程 25914 没有得到票据。因为在该进程出现时，所有的票据都已经被占用了，而之前进程 25915 却得到了一张票据。如果运行该程序多次，可能会看到不同的结果。

13.6.4　进一步的工作

版本 1 的许可证服务器可以很好地工作了：服务器处理请求并且维持持有票据的进程列表。客户可以从服务器得到票据。现在为止一切都正常。看起来这非常理想。不过现实世界并不总这样美好，软件及其使用者并不完全是你所期望的那样。可能出现什么错误呢？该如何处理这些错误呢？

13.7　处理现实的问题

这里的许可证服务器能很好地工作，前提是所有的进程是正常工作的。有时，软件可能运行出错。如果 SuperSleep 程序被另外一个用户杀死了，或者程序发生段存取错误而被内核杀死了，该如何呢？对于它所占用的票据该如何处理呢？如果服务器崩溃了怎么呢？在服务器重启后又将发生什么呢？

现实世界中的程序必须能够处理异常崩溃。这里考虑两种情形：客户端崩溃和服务器崩溃。

13.7.1　处理客户端崩溃

如果客户端崩溃，客户将不会归还票据，如图 13.8 所示。

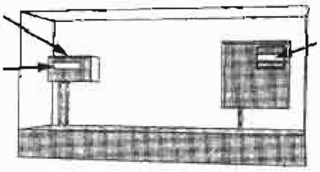


图 13.8　客户不归还票据

在出租车公司里，雇员可能辞职、被解聘、回家或死亡，但仍持有公司轿车的钥匙。这些对公司将造成什么影响呢？签出列表指明票据仍被占用。其他进程就不能得到该票据。但是，如果有足够多的进程崩溃，签出列表就可能虽然满了，但此时却没有运行的客户程序。

钥匙管理员通过给持有钥匙的人打电话，就可以收回钥匙。他可以定期地浏览签出列表，然后给每个司机打电话，“你仍在使用车吗？”。如果无人响应，管理员把他的名字从签出列表中划去。管理员检查的频率越高，签出列表的精确性就越高。

许可证服务器可以使用相同的技术，定期检查票据数组，确认其中的每个进程是否还活着？如果某个进程已经不存在了，服务器可以把该进程从数组中去除，释放其占用的票据。检查程序运行的越频繁，数组的精确性越高。

1. 收回丢失的票据：调度

服务器中如何增加收回票据的代码？如何调用这些代码？服务器必须实现两个独立的操作：等待客户的请求，同时周期性地收回丢失的票据。而调度行为是简单的：只要使用 alarm 和 signal 技术来周期地调用一个函数。在前面章节的移动文字例子中，使用了这种技术。修订的程序流程如图 13.9 所示。

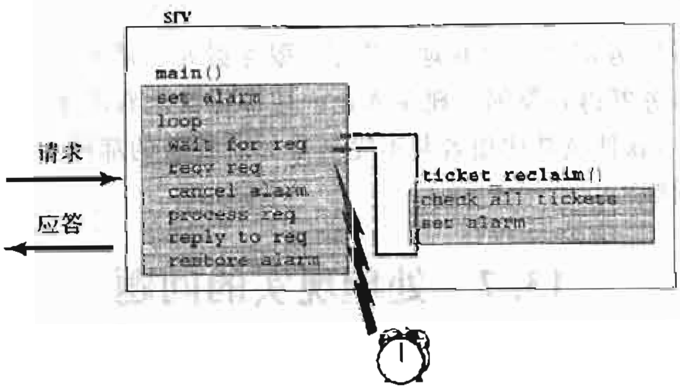


图 13.9 使用 alarm 来调度票据消除程序

在设计需同时处理两件事情的程序时，必定要考虑函数之间的冲突。如果服务器正在处理客户请求的同时，被 SIGALAM 信号触发调用回收丢失的票据的函数，会产生问题吗？这两个处理函数共享变量或者数据结构吗？显然是的，发放票据需要修改签出列表，而收回票据也需要修改签出列表。这种冲突可能会破坏数据的一致性吗？该问题留作课后练习。考虑到安全性，在处理请求的时候关闭 alarm。

2. 收回丢失的票据：编程

服务器希望能回收已经不存在进程的票据。那么如何判断进程是否还活着呢？可以使用 popen 来运行 ps，然后从 ps 的输出中查找 PID，以确定持有票据的 PID 是否存在。另一种快速简洁的方法是使用 kill 系统调用的特殊功能。

可以通过给进程发送编号为 0 的信号以确定它是否存在。如果进程不存在，内核将不会发送信号，而是返回错误并设置 errno 为 ESRCH。在 ticket_reclaim 中使用了该特征，该函数在 lserv_funcs2. c 文件中：

```
#define RECLAIM_INTERVAL 60      /* reclaim every 60 seconds */
/**********************************************************************
 * ticket_reclaim
 * go through all tickets and reclaim ones belonging to dead processes
 * Results: none
 */
void ticket_reclaim()
{
```

```
    int i;
    char tick[BUFSIZ];

    for(i = 0; i < MAXUSERS; i++ ) {
        if((ticket_array[i] != TICKET_AVAIL) &&
        (kill(ticket_array[i], 0) == -1) && (errno == ESRCH)) {
            /* Process is gone - free up slot */
            sprintf(tick, "%d.%d", ticket_array[i],i);
            narrate("freeing", tick, NULL);
            ticket_array[i] = TICKET_AVAIL;
            num_tickets_out--;
        }
    }
    alarm(RECLAIM_INTERVAL);        /* reset alarm clock */
}
```

接下来，在 main 函数中增加调度回收票据的函数，并在正常操作中关闭 alarm。修改过的 main 在文件 lserv2.c 中：

```
int main(int ac, char *av[])
{
    struct sockaddr client_addr;
    socklen_t addrlen = sizeof(client_addr);
    char buf[MSGLEN];
    int ret, sock;
    void ticket_reclaim();              /* version 2 addition */
    unsigned time_left;

    sock = setup();
    signal(SIGALRM, ticket_reclaim);    /* run ticket reclaimer */
    alarm(RECLAIM_INTERVAL);            /* after this delay */

    while(1) {
    addrlen = sizeof(client_addr);
    ret = recvfrom(sock,buf,MSGLEN,0,&client_addr,&addrlen);
    if ( ret != -1 ){
     buf[ret] = '\0';
     narrate("GOT:", buf, &client_addr);
     time_left = alarm(0);
     handle_request(buf,&client_addr,addrlen);
     alarm(time_left);
    }
    else if ( errno != EINTR )
     perror("recvfrom");
    }
}
```

通过上面的修改，许可证服务器就可以周期性地检查票据了。确实需要这样周期性地检查吗？为什么不只在票据列表满了并且有客户的请求被拒绝的时候检查呢？这样会更好吗？

13.7.2　处理服务器崩溃

服务器崩溃通常有两个严重的后果。首先，签出列表丢失，失去进程持有票据的记录。其次，新客户不可以再运行，因为分发许可证的程序已经不存在。最简单的解决方法是重新启动服务器，如图 13.10 所示。

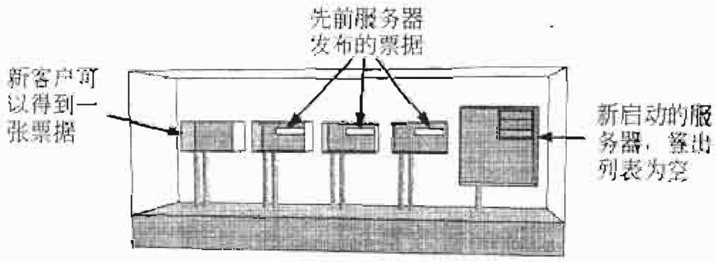


图 13.10　服务器在崩溃后重启

重启服务器使得新的客户可以运行，但会带来两个新的问题。

首先，重启服务器的票据数组是空的；服务器含有新的未被取走的票据列表。崩溃的服务器的票据数组可能已经满了，而重启服务器仍给其他客户发送许可。这样重复地关闭服务器，再重启服务器就像印钞机一样可以产生更多的票据。

其次，持有旧的服务器的票据的客户在归还票据时，将会被认为是伪造票据。

1. 票据验证

上述问题的一个解决方法是票据验证。票据验证表示每个客户周期性地向服务器发送票据的副本。客户发送数据包，对服务器说："这是我的票据。是合法的吗？"，如图13.11所示。

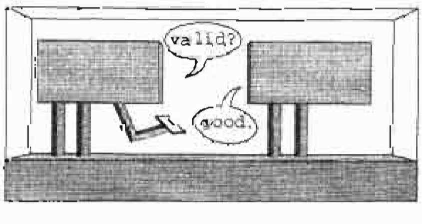


图 13.11　客户验证票据

票据含有数组编号和 PID。服务器检查签出列表。如果该项为空，服务器可以认为该票据是自己先前的实例赋予的。服务器将会把该票据加到列表中。逐步地，客户提供票据来验证，签出列表被重新填入。

服务器重建签出列表解决了表丢失问题，但可能导致其他问题。如果一个新的客户在表重建好之前，请求票据，服务器可能分发一个已经给予其他客户的票据给该客户。当持有旧的票据的客户对其票据进行验证时，服务器会拒绝它。

另一种方法是服务器拒绝表中没有的所有的票据。持有被拒票据的客户尝试再申请新的。这种方法是否更好？

2. 协议中增加验证

票据验证是协议中的一个新的事务：

```
CLIENT:VALD tickid
SERVER:GOOD or FAIL invalid ticket
```

这里必须改变客户和服务器以支持验证。

3. 客户端增加验证

客户端增加验证，需要编写一个函数并在主函数中调用它，其流程如图 13.12 所示。

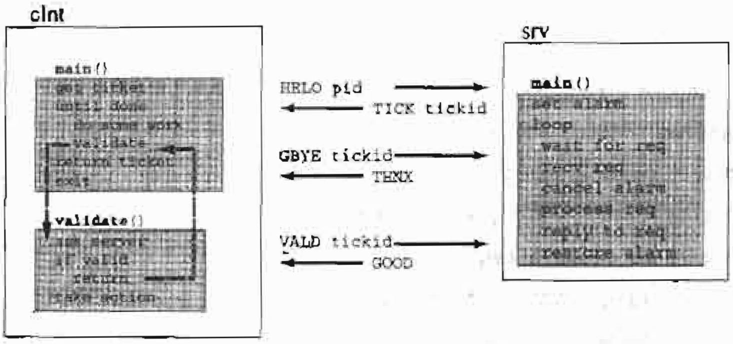


图 13.12　客户定期验证票据

客户可以根据系统的需要以一定的间隔定期验证票据，可以设置一个计时器来定期验证。如果客户是一个电子制表程序，验证可以在一定量的计算结束后进行。一个许可证服务器会响应验证请求。

SuperSleep 客户程序可以把 10 秒的睡眠时间分割成两个 5 秒，在这期间进行验证。这作为课后练习。

4. 服务器端增加票据验证

服务器端增加验证需要做两处改动。改动后的程序位于新的文件 lserv_funcs2.c 中。首先，增加一个函数来验证票据：

```
/*********************************************************************
 * do_validate
 * Validate client's ticket
 * IN msg_p      message received from client
 * Results: ptr to response
 * NOTE: return is in static buffer overwritten by each call.
 */
static char * do_validate(char * msg)
{
    int pid, slot;           /* components of ticket */

    /* msg looks like VALD pid.slot - parse it and validate */
```

```
    if (sscanf(msg + 5, "%d.%d",&pid,&slot) == 2 && ticket_array[slot] == pid)
            return("GOOD Valid ticket");

    /* bad ticket */
    narrate("Bogus ticket", msg + 5, NULL);
    return("FAIL invalid ticket");
}
```

其次，handle_request 中增加更多的判断。

```
handle_request(char * req,struct sockaddr_in * client, socklen_t addlen)
{
    char * response;
    int ret;

    /* act and compose a response */
    if ( strncmp(req, "HELO", 4) == 0 )
        response = do_hello(req);
    else if ( strncmp(req, "GBYE", 4) == 0 )
        response = do_goodbye(req);
    else if ( strncmp(req, "VALD", 4) == 0 )
        response = do_validate(req);
    else
        response = "FAIL invalid request";

    /* send the response to the client */
    narrate("SAID:", response, client);
    ret = sendto(sd, response, strlen(response),0, client, addlen);
    if ( ret == -1 )
        perror("SERVER sendto failed");
}
```

13.7.3　测试版本 2

现在可以编译和测试新版本的客户和服务器了。测试包含了杀死客户和服务器以及重启客户和服务器。试观察输出中的进程 ID 和消息。为了便于测试，客户睡眠时间为两个 15 秒的间隔，服务器每 5 秒尝试回收票据。结果如下：

```
$ cc lserv2.c lserv_funcs2.c dgram.c -o lserv2
$ cc lclnt2.c lclnt_funcs2.c dgram.c -o lclnt2
$ ./lserv2&                            # 启动 1 个服务器
[1]30804
$ ./lclnt2 & ./lclnt2 & ./lclnt2 &     # 启动 3 个客户端
[2]30805
[3]30806
```

```
[4]30807
$               SERVER: GOT: HELO 30805 (10.200.75.200:1085)
                SERVER:SAID:TICK 30805.0 (10.200.75.200:1085)
CLIENT [30805]:got ticket 30805.0
SuperSleep version 1.0 Running - Licensed Software
                SERVER: GOT: HELO 30806 (10.200.75.200:1086)
                SERVER:SAID:TICK 30806.1 (10.200.75.200:1086)
CLIENT [30806]:got ticket 30806.1
SuperSleep version 1.0 Running - Licensed Software
                SERVER: GOT: HELO 30807 (10.200.75.200:1087)
                SERVER:SAID:TICK 30807.2 (10.200.75.200:1087)
CLIENT [30807]:got ticket 30807.2
SuperSleep version 1.0 Running - Licensed Software
$ kill 30806   #kill 一个客户
[3]- Terminated        ./lclnt2
                SERVER:freeing 30806.1
                SERVER: GOT: VALD 30805.0 (10.200.75.200:1085)
                SERVER: SAID: GOOD Valid ticket (10.200.75.200:1085)
CLIENT [30805]:Validated ticket: GOOD Valid ticket
                SERVER: GOT : VALD 30807.2 (10.200.75.200:1087)
                SERVER: SAID: GOOD Valid ticket (10.200.75.200:1087)
CLIENT [30807]: Validated ticket: GOOD Valid ticket
$ kill 30804   #kill 服务器
[1]Terminated        ./lserv2
$ ./lserv2&   #启动新的服务器
[5]30808
$
                SERVER:GOT: GBYE 30805.0 (10.200.75.200:1085)
                SERVER: Bogus ticket 30805.0
                SERVER: SAID: FAIL invalid ticket (10.200.75.200:1085)
CLIETN[30805]:release failed invalid ticket
                SERVER:GOT: GBYE 30807.2 (10.200.75.200:1087)
                SERVER: Bogus ticket 30807.2
                SERVER: SAID: FAIL invalid ticket (10.200.75.200:1087)
CLIETN[30807]:release failed invalid ticket
$ ./lclnt2   #启动一个新的客户
                SERVER: GOT: HELO 30809 (10.200.75.200:10857
                SERVER:SAID:TICK 30809.0 (10.200.75.200:1087)
CLIENT [30809]:got ticket 30809.0
SuperSleep version 1.0 Running - Licensed Software
                SERVER: GOT : VALD 30809.0 (10.200.75.200:1087)
                SERVER: SAID: GOOD Valid ticket (10.200.75.200:1087)
CLIENT [30809]: Validated ticket: GOOD Valid ticket
                SERVER:GOT: GBYE 30809.0 (10.200.75.200:1087)
```

```
                SERVER: SAID: THNX See ya! (10.200.75.200:1087)
CLIENT [30809]: release ticket OK
[2] Done              ./lclnt2
[4]- Done             ./lclnt2
$ ps
PID     TTY     TIME      CMD
23509   pts/3   00:00:00  bash
30808   pts/3   00:00:00  lserv2
30810   pts/3   00:00:00  ps
$
```

看起来还不错。不妨试一下上述程序，观察其中的交互过程。

13.8　分布式许可证服务器

许可证服务器和被许可程序通过 socket 进行通信，socket 可以连接不同主机上的进程。理论上，与 Web 服务器和客户运行在不同主机上类似，这里的客户可以运行在一台机器上，而服务器运行在另一台机器上。当它们运行在不同的机器上时，会有问题吗？是的。

• 问题 1：重复的进程 ID

进程 ID 在一台机器上是惟一的，但在不同主机上的进程可能拥有相同的进程 ID。图 13.13 说明的情况不含有任何错误且很常见。

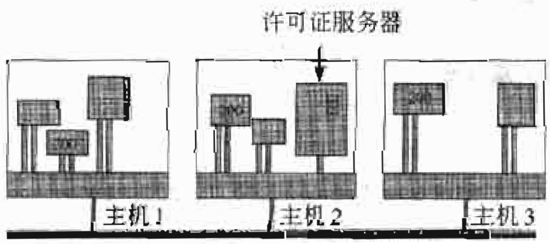


图 13.13　跨网络 PID 不惟一

存放票据的表中含有 PID 和票据编号。在图 13.13 的情况中，许可证服务将认为给同一个进程分配了 3 张票。每个进程只要一张票就可以运行，所以这是错误的。请求更多的票据，可以被认为是客户端的一个漏洞。

这里可以扩展票据表项的格式和内容，使其包含标识运行程序的主机从而解决重复 PID 问题。

• 问题 2：回收票据

服务器通过调用 kill(pid,0)命令向客户回收票据。kill(pid, 0)发送信号 0 给持有票据的进程。通过修订过的票据表，服务器现在可以知道客户运行在哪台主机上。

但是，服务器不能给其他机器上的进程发送信号，如图 13.14 所示。如果许可证服务器想给主机 3 上的进程发送信号，服务器必须产生主机 3 上的请求。

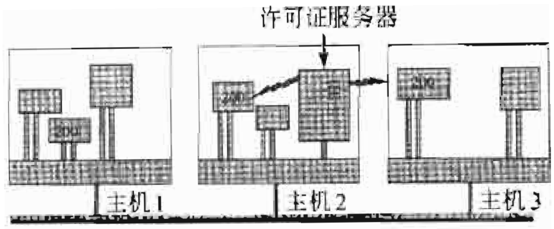


图 13.14 进程不能给其他主机发送信号

为什么不在每台机器上都运行一个服务器的实例？如图 13.15 所示，每个本地的服务器可以监控丢失的票据。

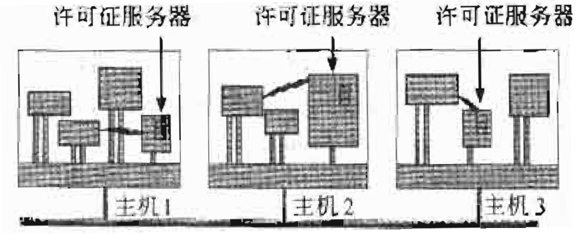


图 13.15 运行本地复制的 lserv

本地服务器解决了向主机发送信号的问题，但是又带来新的问题：哪个服务器发布票据？主服务器如何和本地服务器通信？客户把票据发给谁验证？

• 问题 3：主机崩溃

如果其中的一台机器停止运行，将会发生什么？主服务器如果还在运行的话，如何收回票据？客户端程序如何验证票据？如果运行主服务器的主机停止运行，谁来分发票据？

如何建立一个分布式许可证系统来同时支持多台机器？这里有 3 种方法。试考虑它们的设计细节以及优缺点，并考虑当客户、服务器、计算机或者网络崩溃时每个方案的后果。

• 方法 1：客户端服务器和中央服务器通信

每台机器都有一个本地服务器，就像本文编写的那样。每个客户跟本地的服务器通信。本地服务器把请求转发给中央服务器。中央服务器返回票据或者拒绝。本地服务器记录并把应答转发给客户。本地服务器也可以强制执行一些对本地客户的限制，例如该机器上可以运行的程序实例数，或者程序可以运行的时刻。

• 方法 2：每个客户都和中央服务器通信

客户直接给特定主机上的服务器发送请求。本地服务器运行在每个主机上，但这些服务器不和客户通信，它们在重新声明票据的时候作为中央服务器的代理。

• 方法 3：客户服务器和客户服务器通信

每台机器都有本地服务器，每个客户跟本地服务器通信。没有中央服务器。所有的本地服务器间互相通信。每次一个客户请求时，本地服务器询问其他所有的服务器目前已经用掉了多少张票。如果所用票据数小于所允许的总数，本地服务器分配一张票据给客户。

13.9 Unix 域 socket

本章的许可证服务器中的 socket 使用标准的主机 ID 和端口号地址系统。使用这些 Internet 地址，服务器可以接收本地的乃至更大网络上机器的客户请求。

在上述的两种分布式许可证服务器模型中，客户只需和同一台主机上的服务器通信。socket 只能用在仅限于主机内部的通信中吗？

13.9.1 文件名作为 socket 地址

有两种连接：流连接和数据报连接，也有两种 socket 地址：Internet 地址和本地地址。Internet 地址包含主机 ID 和端口号。本地地址通常叫做 Unix 域地址，它是一个文件名（例如/dev/log、/dev/printer 或/tmp/lserversock），没有主机和端口号。

两个 socket 名/dev/log 和/dev/printer 被用在很多 Unix 系统中。/dev/log 被 syslogd 服务器所使用。想要记录日志的程序只要给地址为/dev/log 的 socket 发送数据报就行了。地址/dev/printer 被一些打印系统使用。

13.9.2 使用 Unix 域 socket 编程

为了学习 Unix 域 socket 的客户/服务器编程，这里编写了一个日志系统。文件 wtmp 就是日志系统的一个实例。文件 wtmp 记录系统的所有登录、退出以及连接。日志系统被保证安全和系统维护的程序所使用，用来记录一些可疑行为。日志服务器是一个抄写员；客户发送信息给服务器，服务器将这些信息保存到只有自己可以修改的文件中。日志服务器可以用任何格式保存该文件，客户并不知道这些细节。

这里的日志服务器使用 Unix 域 socket 地址。只有同一台主机上的客户才能发消息给它。下面是客户和服务器的代码。服务器先创建 socket，然后绑定地址：

```
/**************************************************************
 * logfiled.c - a simple logfile server using Unix Domain Datagram Sockets
 * usage: logfiled >>logfilename
 */

#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <sys/un.h>
#include  <time.h>

#define MSGLEN 512
#define oops(m,x) { perror(m); exit(x); }
#define SOCKNAME "/tmp/logfilesock"

int main(int ac, char * av[])
{
```

```
    int         sock;                       /* read messages here */
    struct      sockaddr_un addr;           /* this is its address */
    socklen_t   addrlen;
    char        msg[MSGLEN];
    int         l;
    char        sockname[] = SOCKNAME ;
    time_t      now;
    int         msgnum = 0;
    char        *timestr;

    /* build an address */
    addr.sun_family = AF_UNIX;              /* note AF_UNIX */
    strcpy(addr.sun_path, sockname);        /* filename is address */
    addrlen = strlen(sockname) + sizeof(addr.sun_family);

    sock = socket(PF_UNIX, SOCK_DGRAM, 0);  /* note PF_UNIX */
    if ( sock == -1 )
        oops("socket",2);

    /* bind the address */
    if ( bind(sock, (struct sockaddr *) &addr, addrlen) == -1 )
        oops("bind", 3);

    /* read and write */
    while(1)
    {
        l = read(sock, msg, MSGLEN);        /* read works for DGRAM */
        msg[l] = '\0';                      /* make it a string */
        time(&now);
        timestr = ctime(&now);
        timestr[strlen(timestr)-1] = '\0';  /* chop newline */

        printf("[%5d] %s %s\n", msgnum++, timestr, msg);
        fflush(stdout);
    }
}
```

这里仍使用 socket 和 bind 来创建服务器 socket。socket 的类型是 SOCK_DGRAM，地址族是 PF_UNIX①。socket 地址是文件名。使用 read 而不是 recvfrom，因为不需要应答。下面是客户端程序：

```
/****************************************************************
 * logfilec.c - logfile client - send messages to the logfile server
 * usage: logfilec "a message here"
```

① PF_LOCAL 也许能替代 PF_UNIX。

```
 */

#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <sys/un.h>

#define   SOCKET       "/tmp/logfilesock"
#define   oops(m,x)    { perror(m); exit(x); }

main(int ac, char *av[])
{
    int              sock;
    struct sockaddr_un addr;
    socklen_t        addrlen;
    char             sockname[] = SOCKET ;
    char             *msg = av[1];

    if ( ac != 2 ){
        fprintf(stderr,"usage: logfilec 'message'\n");
        exit(1);
    }
    sock = socket(PF_UNIX, SOCK_DGRAM, 0);
    if ( sock == -1 )
        oops("socket",2);

    addr.sun_family = AF_UNIX;
    strcpy(addr.sun_path, sockname);
    addrlen = strlen(sockname) + sizeof(addr.sun_family) ;

    if ( sendto(sock,msg, strlen(msg), 0, &addr, addrlen) == -1 )
        oops("sendto",3);
}
```

这里使用 socket 函数来创建 socket，并且使用 sendto 发送消息。服务器接收消息，然后打印消息。消息前面是消息编号和时间。

下面是测试的情形：

```
$ cc logfiled.c -o logfiled
$ ./logfiled >>visitorlog&
1500
$ cc logfilec.c -o logfilec
$ ./logfilec 'Nice system. Swell software!'
$ ./logfilec "Testing this log thing."
$ ./logfilec "Can you read this?"
$ cat vistorlog
[   0] Mon Aug 20 18:25:34 2001 Nice system. Swell software!
```

```
[  1] Mon Aug 20 18:25:44 2001 Testing this log thing.
[  2] Mon Aug 20 18:25:48 2001 Can you read this?
```

上述两个程序展示了如何使用 Unix 域 socket 以及日志服务器的基本思想。程序的另一个特征是实现了自动追加功能，但没有使用 O_APPEND。服务器接收消息，然后把消息追加到文件末尾。即使有多个客户同时发送消息，底层的 socket 机制会负责消息的序列化。

13.10　小结：socket 和服务器

socket 对于进程间的数据通信是强大的、万能的工具。这里学习了两种 socket 和两种 socket 地址。

	域	
socket	**PF_INET**	**PF_UNIX**
SOCK_STREAM	连接的，跨机器	连接的，本地
SOCK_DGRAM	数据报，跨机器	数据报，本地

在前面的几章中，已经学习了使用这 4 种组合中的 3 种 socket 的项目。当考虑 Unix 的网络编程和准备设计自己的项目时，可以考虑使用这张表格中的技术。根据发送消息的类型以及消息发送的距离来选择最佳的技术。

小　　结

1. 主要内容

- 数据报是从一个 socket 发送到另一个 socket 的短消息。数据报 socket 是不连接的，每个消息包含有目的地址。数据报(UDP)socket 更加简单、快速，给系统增加的负荷更小。
- 许可证服务器是用来对被许可程序实施许可证验证规则的。许可证服务器发布许可，以短消息的形式发送给客户。
- 许可证服务器必须记住哪个进程使用了哪张票据，必须维持一个内部的数据库。因此，许可证服务器不同于本书的 Web 服务器。
- 记录系统状态的服务器必须设计成可以处理服务器和客户端的崩溃事件。
- 有些许可证服务器为一个网络上的多个机器提供服务。有几种设计方法，各有优缺点。
- socket 可以有两种类型的地址：网络或本地。本地的 socket 地址叫做 Unix 域 socket 或名字 socket。这种 socket 使用文件名作为地址，只能在一台机器上交互数据。

2. 下一步的工作

本章学习了两种服务器处理多个请求的方法。许可证服务器接收数据报请求，并且一次一条地返回消息。Web 服务器接收数据流消息，并且使用 fork 同时应答所有请求。服务

器有另外的选择：一个单一进程可以使用线程的技术同时运行几个函数。下一章将学习有关线程的思想和技术。

3. 习题

13.1 在使用流 socket 的例子中，服务器给客户应答时，并没有使用客户端的地址。服务器是如何知道给哪个地址的客户发消息的？

13.2 进程 25915 是如何战胜进程 25914 得到票据的？考虑每个客户进程的创建和服务器端请求到达的操作序列。在多任务系统中，进程依次运行。进程在哪儿可以被中断从而得到测试的结果？

13.3 如何使用一个许可证服务器处理 2 个或更多的程序的请求？一种方法是修改协议使得每个请求包含所要运行的程序名。描述可以支持多程序协议的数据结构和程序逻辑。

13.4 当服务器正在处理客户请求时，如果函数 ticket_reclaim 被调用，是否会对票据列表存在潜在的破坏？考虑函数中每个更改数组和计数器的地方。在哪一点上数组的状态和计数器的值不一致？处理函数是如何修改数组和计数器的？这些值的一个未预测的改变对常规处理函数有何影响？

13.5 当进程被创建时，进程 ID 被分配给进程。考虑下面的时间序列：一个进程 ID 为 777 的客户得到票据后崩溃了。不久，一个不同的用户运行程序创建了一个新的进程，进程 ID 恰好也是 777。当 ticket_reclaim 程序运行时，它发现了进程 777。即使现在编号为 777 的进程是一个不相关的程序，并且不持有票据，分配给原来进程 777 的票据也不能被回收。当这种情形出现时，该如何处理？如何避免这类事件的发生。

13.6 一种防止票据数组丢失的方法是服务器将数据写到磁盘文件中。如果要适应这种备份机制，该如何改变服务器？假设客户会故意杀死服务器以得到更多的票据，那么这里的文件备份机制在此种情形下该如何工作？

13.7 持有早期版本票据的客户在等待了较长的时间后来验证票据，可能发现已经没有更多的可用票据了。为这种情形设计一种应答，使得客户不允许继续运行，因为这将破坏最大同时运行进程的数目，但要求客户不能突然退出。

13.8 参照本章中列出的问题，比较三种分布式许可证服务器模型。

13.9 使用 socket 时，可以用 write 和 sendto 来发送数据。阅读 send 和 sendmsg 的帮助手册，后面两个函数与前面的有什么区别？

13.10 轿车管理系统与数据报不只是比喻。设想每部轿车都装有 GPS 设备。一个计算机可以通过 modem 连接到 Internet，使得可以定位轿车的位置。设想轿车的启动不是受钥匙控制，而是通过一个磁卡的登录来控制。设计一个允许雇员登

录到系统来使用轿车，并且管理员可以跟踪司机行驶路线以及定位轿车位置的系统。

4. 编程练习

13.11 更改程序 dgrecv.c，要求不仅打印出发送者的地址和消息接收的时间，还要打印消息编号。消息标号从 0 开始。希望得到的输出如下：

```
dgrecv:got a message : testing 123
from: 10.200.75.200:1041
at :Sun Aug 19 10:22:27 EDT 2001
msg# :23
```

13.12 为 dgsend.c 增加客户程序 dgrecv2.c。

13.13 许可证服务器在一张表中存有客户拥有票据的信息。如果想要服务器打印这些信息，该如何操作？看到表的信息有助于排错或测试服务器。一种标准的和服务器通信技术是使用信号。

更改程序 lserv1 使得它在接收信号 SIGHUP 时，打印出表的内容。可以通过命令 kill －HUP serverpid 来测试该特征。

13.14 更改许可证服务器使得它只在一个客户的请求被拒绝时，才调用 ticket_reclaim 程序。该方法的优缺点各是什么？

13.15 更改 lclnt2.c 程序，使它睡眠 5 秒钟后验证票据。如果票据合法，客户再睡眠 5 秒钟，然后退出并归还票据。如果票据不合法，客户尝试请求另一个票据。如果成功，继续正常操作。如果失败，告诉用户许可证服务器出错然后再退出。

13.16 更改前面章节中编写的 shell 程序和 bounce 程序，使用许可证服务器。在哪增加票据验证过程？当服务器崩溃时，票据不可用了，该和用户说什么？

13.17 更改客户服务器代码使得票据包含有主机 IP 地址。如何改变票据列表？确信对验证函数也做了改变。

13.18 实现三种分布式许可证控制模型的一种。

13.19 日志系统的一个问题是其中的消息都是匿名的，更改该系统使得消息包含发送消息用户的用户名。

13.20 在日志服务器中使用 read。编写两个新版本的服务器，一个使用 recvfrom，另一个使用 recv。这些获取数据的方法有何不同？更详细的信息请阅读帮助手册。

13.21 如果许可证服务器和客户需要使用 Unix 域 socket，需要做何改变？解释客户为什么要使用 bind。

13.22　在第 1 章中，讨论了一个 Internet 桥牌游戏。在任何的分布式扑克游戏中，软件必须模拟一个扑克牌栈，以确保两个客户不会持有相同的扑克。

编写一对程序 cardd 和 cardc，使用数据报来处理一桌的扑克牌。启动时服务先洗牌，接下来客户从命令行启动，就可以从服务器获取扑克牌。例程运行如下：

```
$ cardc get 5
4D AH 2D TD KC
```

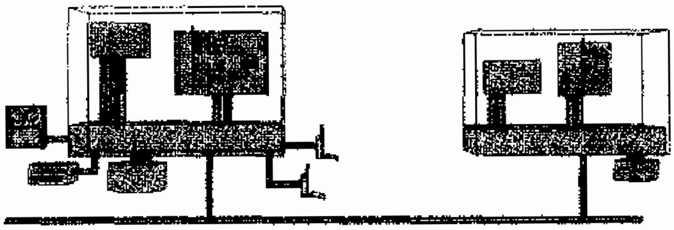
结果显示客户得到了 5 张牌，一张方块 4，一张红桃 A，一张方块 2，一张方块 10，一张王。确保程序不会将相同的扑克牌发给两个用户机，并且协议要有途径表明 dealer 何时发完牌。考虑一下还有其他什么有用的事务可以作为协议的补充？

5. 项目

基于本章的内容，可以学习编写下面的 Unix 程序：

talk、rwho、流媒体服务器

第 14 章　线程机制：并发函数的使用



概念与技巧

- 程序的执行路线
- 多线程程序
- 创建及销毁线程
- 使用互斥锁机制保证线程间数据的安全共享
- 使用条件变量同步线程间的数据传输
- 传递多个参数给线程

相关的系统调用与函数

- pthread_create、pthread_join
- pthread_mutex_lock、pthread_mutex_unlock
- pthread_cond_wait、pthread_cond_signal

14.1　同一时刻完成多项任务

不知道你是否经常会被一些充满了闪烁、跳动或者旋转图片的网页弄得很不舒服，不过我确实不喜欢它们。然而尽管这些网页让人很烦，它们却引发了一个技术问题：一个程序如何才能在同一时刻完成多个任务呢？

动画图片并不是惟一可以完成并发功能的 Web 程序。想想看日常生活中的浏览器，它可以从不同的服务器上下载并且解压缩图片。浏览器并行地完成这么多的任务，而非一项接一项地做，那么它又是如何同时下载并解压这些图片的？

多任务系统的问题在本书中早已介绍过了。视频游戏那一章，使用计时器和两个计数器在两维空间中控制图片的动作。其他的章节也曾用 fork 和 exec 创建新进程的方法处理并发程序，从而实现一个 Web 服务器的功能。那么这里为何不用这种方法呢？

书中曾使用 fork 和 exec 同时运行多个程序。如果希望同时运行几个函数，或者同时对一个函数调用很多次，那该怎么办呢？

本章将介绍线程。线程相对于函数就类似于进程相对于程序，后者为前者提供了运行

环境。很多函数可以同时运行，但它们都在相同的进程中。

本章主要的项目是完成一个程序让文本框中出现不断移动的文字。这里将先修改以前的那个 Web 服务器程序，让它不启动新的进程即可处理访问目录列表以及文件内容的并发请求。

14.2 函数的执行路线

什么是线程？有何用？又如何去创建它呢？首先看下面的一个传统的程序，在此程序中，指令一条接一条的顺序执行。然后，只要将此程序做两个小的改动，即可让程序并行的执行两个函数。

14.2.1 一个单线程程序

```
/* hello_single.c - a single threaded hello world program */

#include <stdio.h>
#define NUM 5

main()
{
    void print_msg(char * );

    print_msg(("hello");
    print_msg(("world\n");
}
void print_msg(char * m)
{
    int i;
    for (i = 0 ; i<NUM ; i ++ ){
        printf(("%s", m);
        fflush(stdout);
        sleep(1);
    }
}
```

在 hello_single.c 中，main 函数顺序地调用了两个函数。每个函数执行了一个循环。下面的输出结果反映了程序的控制流：

```
$ cc hello_single.c -o hello_single
$ ./hello_single
hellohellohellohellohelloworld
world
world
world
world
$
```

上面的每一行输出之间都有一个 1 秒钟的时延，程序共运行了 10 秒。图 14.1 展示了此程序的执行流程。

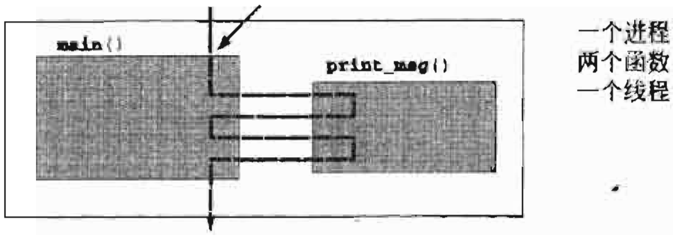


图 14.1　单执行路径

首先，执行路径进入 main 函数，然后进入函数 print_msg。接着，从 print_msg 中返回到 main，之后再一次进入 print_msg 函数进行第二次的函数调用。所有指令执行完毕后，从 main 中返回。

不间断地跟踪指令执行的路径在这里被称做执行路线(thread of execution)。传统的程序只有一条单独的执行路线。就算是包含有 goto 语句以及递归子程序的程序也只有一条执行路线，尽管这条路线有时有些弯弯绕绕。

14.2.2　一个多线程程序

如果想同时执行两个对于 print_msg 函数的调用，就像使用 fork 建立两个新的进程一样，那该怎么办呢？这种思想清楚地体现在图 14.2 中。

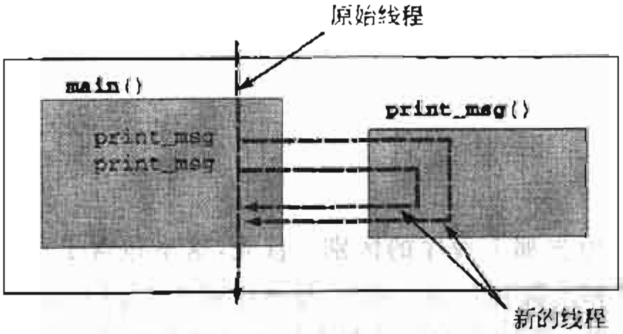


图 14.2　多执行路径的程序

首先，一条执行线路进入 main 函数。初始的线路新建了一条新的执行线路来运行函数 print_msg。初始线路继续执行下一条指令从而新建了另一条线路来对 print_msg 函数进行第二次的调用。最后，初始线路等待两条新的线路加入自己，再从 main 函数中返回。

人们无时无刻不在进行着这种多线程的任务管理。如果父母需要做许多琐事，他们通常会带上孩子一起去。父母让一个孩子到杂货铺买牛奶，另一个孩子去图书馆还书。最后等两个孩子都回来之后，大家再一起回家。

一个线程就类似于上例中帮父母做事情的一个孩子。如果想同时完成许多事情，最好

多带几个孩子一起去。类似地，如果一个程序希望同时执行很多函数，它必须创建多个线程。下面的这个程序 hello_multi.c 将图 14.2 的思想实现了。

```
/* hello_multi.c - a multi - threaded hello world program */

#include <stdio.h>
#include <pthread.h>

#define NUM 5

main()

{
    pthread_t t1, t2;      /* two threads */

    void * print_msg(void * );

    pthread_create(&t1, NULL, print_msg, (void * )"hello");
    pthread_create(&t2, NULL, print_msg, (void * )"world\n");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
void * print_msg(void * m)
{
    char * cp = (char * ) m;
    int i;
    for (i = 0 ; i<NUM ; i++ ){
        printf(" %s", m);
        fflush(stdout);
        sleep(1);
    }
    return NULL;
}
```

注意一下此程序与原先那个程序的区别。首先，这里包含了一个头文件 pthread.h，它包含了数据类型的定义和函数的原型。其次，程序中定义了 pthread_t 类型的两个变量 t1 和 t2。这两个线程就类似于上面所说的父母办事时所带的两个孩子。

图 14.2 控制流中的每一个分支点都对应了如下的一行代码：

```
pthread create(&t1,NULL,print_msg,(void* )"hello")
```

此函数调用就类似于父母喊道："嗨，孩子，用参数 hello 来运行函数 print_msg。"上面第一个参数是线程的地址；第二个参数是指向线程属性的指针；第三个参数是所要执行的函数名称；而第四个参数则是指向所要传递给函数的参数的指针。

这条指令使用指定的属性新建了一个线程，而此线程使用参数 hello 来运行函数 print_msg。

```
pthread_create(&t2,NULL,print_msg,(void * )"world\n")
```

此函数也使用默认的属性创建了一个新的线程。新的线程用参数 world\n 来运行函数 print_msg。

而函数 pthread_join(t1,NULL)；类似于父母等待孩子归来一样，main 借助于此函数来等待两个线程执行路线的返回。函数 pthread_join 使用了两个参数。第一个参数是所要等待的线程，而第二个参数则是一个指向返回值的指针。如果此参数为 NULL，表示返回值不被考虑。

pthread_join(t2,NULL)；表示 main 函数等待另一个线程返回。

编译此程序，并运行，输出结果如下：

```
$ cc hello_multi.c - lpthread - o hello_multi
$ ./hello_multi
helloworld
helloworld
helloworld
helloworld
helloworld
$
```

此程序只运行了 5 秒钟，因为两个循环并行的执行。不过对于线程调度的不同可能会导致输出与上面所示的输出不同。大家可能已经注意到，线程的使用是多么的灵活。在这里同时运行了同一个函数的两个不同实例，仅仅参数是不同的。当然同时运行两个不同的函数也是一样的容易。

14.2.3　相关函数小结

pthread_create		
目标	创建一个新的线程	
头文件	#include <pthread.h>	
函数原型	int pthread_create (pthread_t　　* thread, pthread_attr_t　* attr, void　　* (* func)(void *) void　　* arg);	
参数	thread	指向 pthread_t 类型变量的指针
	attr	指向 pthread_attr_t 类型变量的指针，或者为 NULL
	func	指向新线程所运行函数的指针
	arg	传递给 func 的参数
返回值	0	成功返回
	errcode	错误

pthread_create 函数创建了一条新的执行线路，在此新的线程内调用了 func(arg)。新线程的属性由 attr 参数来指定。func 是一个函数，它接收一个指针作为它的参数，并且运行结束后返回一个指针。参数和返回值都被定义为类型为 void * 的指针，以允许它们指向任何类型的值。

如果 attr 值为 NULL，线程使用的是默认的属性。下一章中将讨论线程的属性。pthread_create 如果运行成功返回 0，否则返回一个非零错误代码。

Pthread_join		
目标	等待某线程终止	
头文件	#include <pthread.h>	
函数原型	int pthread_join(pthread_t thread, void ** retval)	
参数	thread	所等待的线程
	retval	指向某存储线程返回值的变量
返回值	0	成功返回
	errcode	错误

pthread_join 使得调用线程挂起直至由 thread 参数指定的线程终止。如果 retval 不是 null，线程的返回值就将存储在由 retval 指向的变量中。

当线程终止时，pthread_join 函数返回 0，如果有错误发生，则返回一个非零错误代码。如果某线程试图等待一个并不存在的线程、多个线程同时等待一个线程返回或者线程试图等待自己都将导致函数返回一个错误代码。

使用线程进行编程就像给一些人赋予不同的任务。如果加强项目管理，保证所有的人都能够按序办事，不和别人冲突，这个项目肯定会提前完成。下面将介绍可以使线程分工合作的技术。

14.3　线程间的分工合作

进程间可以通过管道、socket、信号、退出/等待以及运行环境来进行会话。线程间的通信也很容易。多个线程在一个单独的进程中运行，共享全局变量，因此线程间可以通过设置和读取这些全局变量来进行通信。不过要知道，对共享内存的访问可是线程的一个既有用又极为危险的特性。

14.3.1　例 1：incrprint.c

```
/* incprint.c - one thread increments, the other prints */

#include <stdio.h>
#include <pthread.h>

#define     NUM     5
```

```
int counter = 0;

main()
{
    pthread_t t1;                      /* one thread */
    void      *print_count(void *);    /* its function */
    int       i;

    pthread_create(&t1, NULL, print_count, NULL);
    for ( i = 0 ; i<NUM ; i++ ){
        counter++ ;
        sleep(1);
    }

    pthread_join(t1, NULL);
}
void *print_count(void *m)
{
    int i;
    for (i = 0 ; i<NUM ; i++){
        printf("count = %d\n", counter);
        sleep(1);
    }
    return NULL;
}
```

程序 incprint.c 使用了两个线程。初始线程执行了一个循环来使计数器值每秒钟增 1。初始线程在进入循环之前，创建了一个新的线程。新的线程运行了一个函数来将 counter 的值打印出来。main 函数和 print_count 函数运行在同一个进程中，所以都有对于 counter 的访问权。图 14.3 展示了两个函数和全局变量的内在逻辑。

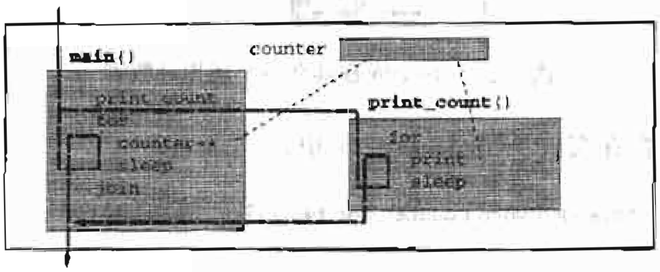


图 14.3 两个线程共享全局变量

当 main 函数改变了 counter 值之后，print_counter 函数立即可以访问到新的值。因此并不需要通过管道或者套接字等方法传送新的值。编译这个程序，然后运行它，结果如下：

```
$ cc incprint.c -lpthread -o incprint
$ ./incprint
```

```
count = 1
count = 2
count = 3
count = 4
count = 5
```

程序显然可以正常工作。一个函数修改了变量，另一个函数读取并显示了变量的值。这个例子展示了如何使运行在不同线程中的函数共享全局变量。不过下个例子还要有趣。

14.3.2 例 2：twordcount.c

很多学生都有这样的经验，对着电脑数自己学期论文的字数以确定字数是不是足够。假设一个学生有一篇 10 页纸的论文，它有两种方法来计算这篇文章的字数。一种是一个字一个字地数了 10 页纸，另一种方法是找 10 个同学来，给每个同学一页纸，让他们分别计算，然后将结果累加起来。显然并行地计算这 10 页纸的字数的方法会快很多。

Unix 平台上的 wc 程序的作用是计算一个或多个文件中的行、单词以及字符个数。不过 wc 是一个典型的单线程程序。怎样来设计一个多线程程序来计数并打印两个文件中的所有字数呢?

- 版本 1：两个线程、一个计数器

第一个版本程序创建分开的线程来对每一个文件进行计算。所有的线程在检查到单词的时候对同一个计数器增值。图 14.4 体现了这个思路。

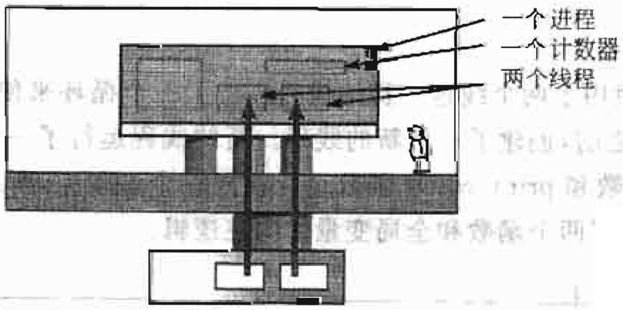


图 14.4 两个线程共享一个通用计数器

此版本的代码包含在文件 twordcount1.c 中：

```
/* twordcount1.c - threaded word counter for two files. Version 1 */

  #include <stdio.h>
  #include <pthread.h>
  #include <ctype.h>

  int total_words ;

  main(int ac, char *av[])
  {
     pthread_t t1, t2;          /* two threads */
```

```
    void    * count_words(void * );

    if ( ac != 3 ){
        printf("usage: %s file1 file2\n", av[0]);
        exit(1);
    }
    total_words = 0;
    pthread_create(&t1, NULL, count_words, (void * ) av[1]);
    pthread_create(&t2, NULL, count_words, (void * ) av[2]);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%5d: total words\n", total_words);
}
void * count_words(void * f)
{
    char * filename = (char * ) f;
    FILE * fp;
    int c, prevc = '\0';

    if ( (fp = fopen(filename, "r")) != NULL ){
        while( ( c = getc(fp))!= EOF ){
            if ( ! isalnum(c) && isalnum(prevc) )
                total_words ++ ;
            prevc = c;
        }
        fclose(fp);
    } else
        perror(filename);
    return NULL;
}
```

函数 count_words 是这样区分单词的：凡是一个非字母或数字的字符跟在字母或数字的后面，那么这个字母或数字就是单词的结尾。当然这种思路忽略了文件的最后一个单词，并且还把“U. S. A. ”看成三个独立的单词。编译此程序并按照如下的方法进行测试：

```
$ cc twordcount1.c -lpthread -o twc1
$ ./twc1 /etc/group /user/dict/words
45614: total words
$ wc -w  /etc/group /usr/dict/words
      58 /etc/group
   45402 /user/dict/words
   45460 total
```

twordcount1 产生的结果与 wc 并不相同，因为两个程序对于单词结尾规则的定义不同。这里还有一个比单词结尾更加微妙的问题：所有线程对同一个计数器进行操作，并且在

同时进行。细心的读者也许会问：这样会不会有问题啊？C 语言并没有指定操作 total_words++ 是如何被计算机执行的。计算机有可能执行的是这样一个操作：

```
total_words = total_words + 1
```

也就是说，程序先将计数器当前的值存入寄存器中，加 1 操作后，再将其恢复到内存中。

那么如果所有线程在同一时刻都使用"取出－加－存储"的序列来完成对计数器的操作，结果会如何呢？

图 14.5 所示的是所有线程取出同样的值，对寄存器增 1，然后再恢复新的值。两次增 1 操作同时发生，但是计数器的值只能一次一次的增加。如何来保证线程间不会互相干扰对方工作呢？下面将采用两种办法进行尝试。

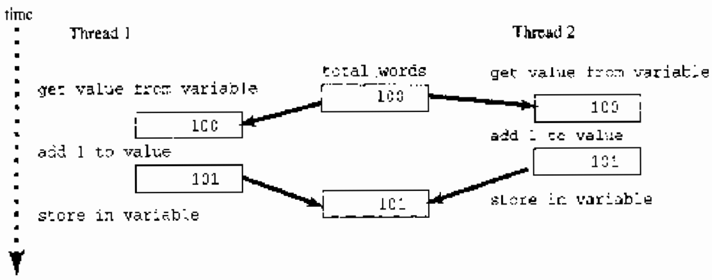


图 14.5　两个线程对同一个计数器进行操作

· 版本 2：两个线程、一个计数器、一个互斥量

大家注意到在机场或者公交车终点站的公共存储柜始终是打开的，除非有人在里面存了东西。当一个人扔了硬币然后拿到了钥匙之后，便没有别人可以再去打开那个柜子了。只有等到这个人归还了钥匙，把柜子打开之后，其他人才可以再去使用。类似地，如果两个线程需要安全地共享一个公共的计数器，它们也需要这样一种方法把变量加锁。

线程系统包含了称为互斥锁的变量，它可以使线程间很好的合作，避免对于变量、函数以及资源的访问冲突。下面的程序 twordcount2.c 将告诉大家如何创建和使用互斥量：

```
/* twordcount2.c - threaded word counter for two files. */
/*                 version 2: uses mutex to lock counter */
#include <stdio.h>
#include <pthread.h>
#include <ctype.h>

int        total_words ;                   /* the counter and its lock */
pthread_mutex_t counter_lock = PTHREAD_MUTEX_INITIALIZER;

main(int ac, char * av[])
{
    pthread_t t1, t2;                      /* two threads */
    void     * count_words(void * );

    if ( ac!= 3 ){
```

```
        printf("usage: %s file1 file2\n", av#[0];
        exit(1);
    }
    total_words =[0];
    pthread_create(&t1, NULL, count_words, (void * ) av[1]);
    pthread_create(&t2, NULL, count_words, (void * ) av[2]);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%5d: total words\n", total_words);
}
void * count_words(void * f)
{
    char * filename = (char * ) f;
    FILE * fp;
    int c, prevc = '\0';

    if ( (fp = fopen(filename, "r"))!= NULL ){
        while( ( c = getc(fp))!= EOF ){
            if ( ! isalnum(c) && isalnum(prevc) ){
                pthread_mutex_lock(&counter_lock);
                total_words++ ;
                pthread_mutex_unlock(&counter_lock);
            }
            prevc = c;
        }
        fclose(fp);
    } else
        perror(filename);
    return NULL;
}
```

此程序的逻辑类似于图 14.6 所示。

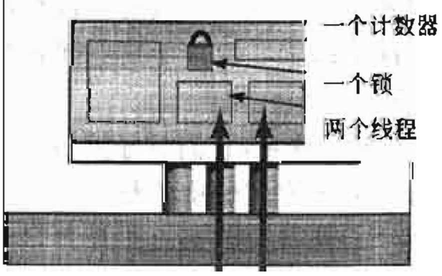


图 14.6 两个线程使用互斥锁来共享计数器

细心的读者可以发现，在原来的程序中仅仅加了三行代码。首先定义了一个 pthread_mutex_t 类型的全局变量 counter_lock，然后赋给它一个初值。另外改动的地方就是把对于 count_words 的操作夹在对两个函数 pthread_mutex_lock 以及 pthread_mutex_unlock 的调用之间。

现在两个线程可以安全地共享计数器了。当一个线程调用 pthread_mutex_lock 的时候，如果另一个线程已经将这个互斥量锁住了，那这个线程只好阻塞等待着这个锁被另一个线程解开后，才可以对计数

器进行操作。每个线程对计数器进行操作后，都将互斥量解锁，然后循环地处理其他数据。

任何数目的线程都可以挂起等待互斥量解锁。当一个线程对互斥量解锁之后，系统就将控制权交给等候的某一线程。如所有线程都按照这种互斥原则进行通信时，互斥量是有用的；然而如果一个线程不遵守这个规则，直接去修改计数器的值的话，程序员也无能为力。

pthread_mutex_lock		
目标	等待互斥锁解开然后再锁住互斥量	
头文件	#include <pthread.h>	
函数原型	int pthread_mutex_lock(pthread_mutex_t * mutex)	
参数	mutex	指向互斥锁对象的指针
返回值	0	成功返回
	errcode	错误

pthread_mutex_lock 函数用来锁住指定的互斥量。如果互斥量是开放的，它被锁住，并只能由调用线程来管理。如果此时互斥量已经被另外的线程锁住，调用线程将挂起等待此互斥量被解锁。如果调用过程中出现错误，函数将返回一个错误代码。

pthread_mutex_unlock		
目标	给互斥量解锁	
头文件	#include <pthread.h>	
函数原型	int pthread_mutex_unlock(pthread_mutex_t * mutex)	
参数	mutex	指向互斥锁对象的指针
返回值	0	成功返回
	errcode	错误

pthread_mutex_unlock 函数给指定的互斥量解锁。如果有线程挂起等待此互斥量，其中的一个线程将获得对互斥锁的控制权。若解锁成功，此函数将返回 0，否则返回一个非零的错误代码。

不过上面讨论的都是正常的情况。如果一个线程试图解锁一个并没有被锁住的互斥量那会怎么样？如果线程企图对一个已经锁住的互斥量加锁呢？又如果一个线程还没有对自己锁住的互斥量解锁就退出了，那么结果又会如何呢？不同的线程系统对于这些问题的处理方法各不相同。详情请参见你所使用的 Unix 的参考手册。

是否需要互斥量？如果多个线程企图在同一时刻修改相同的变量，它们只好使用互斥量来避免访问冲突。然而使用互斥量使得程序运行速度变慢。对所有文件中的每一个单词都需要执行检查，设置以及释放锁的操作，这使得程序效率低下。更加有效的方法是为每个线程设置自己的计数器。

- 版本 3：两个线程、两个计数器、向线程传递多个参数

下一个版本的字数统计程序为每个线程设置了自己的计数器，从而避免了对于互斥量的使用。当线程返回之后，再将这两个计数器的值加起来得到最后的结果。

如何来得到这些线程的计数器？又如何使线程将它们的计数值返回呢？在一个通常的单线程程序中，字数统计函数将得出的字数返回给它的调用函数。线程可以通过调用 pthread_exit 得到返回值，这个返回值又可以通过 pthread_join 的调用被原先的线程得到。详情可以参见手册。下面将使用一个稍微简单一点的方法。

调用线程通过传递给函数一个指向某变量的指针，让函数对此变量进行操作，从而可以避免让线程将值传回。传递指针引发了一个问题：函数 pthread_create 只能允许传递一个参数给函数，而文件名又必须传给函数，那么如何传这个指针呢？办法很简单，只需建一个包含两个成员的结构体，然后将此结构体的地址传给函数即可。

```
/* twordcount3.c - threaded word counter for two files.
 *              Version 3: one counter per file
 */

#include <stdio.h>
#include <pthread.h>
#include <ctype.h>
struct arg_set {                        /* two values in one arg */
        char * fname;                   /* file to examine      */
        int count;                      /* number of words */
};
main(int ac, char *av[])
{
    pthread_t       t1, t2;             /* two threads */
    struct arg_set args1, args2;        /* two argsets */
    void        * count_words(void * );

    if ( ac!= 3 ){
        printf("usage: %s file1 file2\n", av#k[0];
        exit(1);
    }
    args1.fname = av[1];
    args1.count = 0;
    pthread_create(&t1, NULL, count_words, (void * ) &args1);

    args2.fname = av[2];
    args2.count = 0;
    pthread_create(&t2, NULL, count_words, (void * ) &args2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%5d: %s\n", args1.count, av[1]);
    printf("%5d: %s\n", args2.count, av[2]);
    printf("%5d: total words\n", args1.count + args2.count);
}
void * count_words(void * a)
```

```
{
    struct arg_set * args = a;          /* cast arg back to correct type */
    FILE * fp;
    int c, prevc = '\0';

    if ( (fp = fopen(args->fname, "r"))!= NULL ){
        while( ( c = getc(fp))!= EOF ){
            if ( ! isalnum(c) && isalnum(prevc) )
                args->count++
            prevc = c;
        }
        fclose(fp);
    } else
        perror(args->fname);
    return NULL;
}
```

这里通过定义一个以文件名和该文件中字数为成员的结构体解决了同时传递两个参数的问题。main 函数定义了两个这种类型的局部变量，并将这两个变量的地址传给线程（如图 14.7 所示）。传递本地结构体指针的方法既避免了对互斥量的依赖，又消除了全局变量。

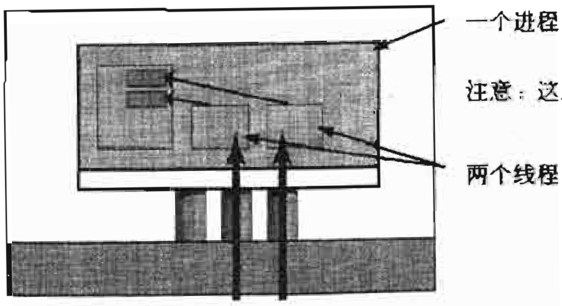


图 14.7　每个线程都拥有一个指向自己结构体的指针

每次调用函数 count_words 之后都会接收到一个指向不同结构体的指针，因此线程从不同的文件中读取信息，并对不同的计数器进行增 1 操作。因为结构体是 main 中的局部变量，所以分配给各计数器的内存空间在 main 函数返回前一直保存着。

14.3.3　线程内部的分工合作：小结

进程的数据空间包含了所有属于它的变量。此进程中运行的所有线程都拥有对这些变量访问的权限。如果这些变量值不变的话，线程可以无误地读取并使用它们的值。

不过如果进程中的任何线程修改了一个变量值，所有使用此变量的线程必须采用某种策略来避免访问冲突。在某一时刻，只有惟一的线程可以对变量进行访问。

字数统计程序的三个不同版本显示了三种不同的方法来进行线程间的变量共享。在 twordcount1. c 中使用的第一种方法，允许线程无任何合作来修改同一个变量。这个程序本身存在着很大问题。

程序 twordcount2. c 中使用了第二种方法。这种方法使用一个互斥对象来保证在某一时刻，只有惟一的一个线程对计数器进行修改。此程序虽然解决了问题，但是由于对设置和释放互斥锁太多的调用导致了系统性能的降低。

twordcount3. c 中使用的第三种方法是一种改进的方法。此方法为每一个线程创建了各自的计数器，这样避免了共享计数器的麻烦。所有的线程不需要再共享一个变量，因此也不用互相合作了。不过尽管如此，每一个单独的线程仍需和原先的线程进行合作。特别地，原线程不应在其他线程返回之前读取它们各自计数器的内容。在这种情况下，原线程使用 pthread_join 函数使自己挂起直到线程已返回。当某一计数线程返回的时候，对于 pthread_join 的调用激活原线程，允许其访问计数器并且也告诉 main 函数该是读取计数器值的时候了。

第三个版本的程序展示了如何传递多个参数给某一线程中的函数。即先创建一个结构类型变量让它包含所有的参数，再将此结构体的地址传给函数。于是线程可读取或修改此结构体中的任意成员变量了。任何访问此结构体的函数都可以看见值的不断变化。当然，如果不止一个线程需要修改这些值的时候，就又得借助于互斥量来避免访问冲突了。

14.4　线程与进程

Unix 从其产生伊始就将进程作为它的重要组成部分而线程是后来才加进去的。进程的概念非常清晰且统一。而线程却有着一系列的起源，它们的属性也各不相同。这里的例子用了一个叫做 POSIX 的线程接口。在这里当然忽略了效率和调度的问题，这些由你所使用的 Unix 版本和线程版本所决定。

进程与线程有根本上的不同。每个进程有其独立的数据空间、文件描述符以及进程的 ID。而线程共享一个数据空间、文件描述符以及进程 ID。下面这些概念对于程序员来说是非常重要的。

(1) 共享数据空间

这里考虑一个在存储器中存储了巨大而复杂的树结构数据库的数据库系统。多个线程可以轻易地读取到这个共享的数据集。客户的多个查询可以由一个进程来实现。如果变量不会被改变，共享这个数据空间不会导致任何问题。

再考虑一个使用 malloc 和 free 系统调用来管理内存的程序。一个线程分配了一块空间存储一个字符串。当此线程做其他事情的时候，另一个线程使用 free 释放了这块空间。那么原先的线程中本来指向此空间的指针现在指向了一块已经被释放的地方，更糟糕的是，这块地方已经被派上别的用处了。

线程机制还会带来内存的囤积。程序员往往因为怕影响了某线程正在使用的内存空间，只分配而不释放存储区域。这直接导致了内存的囤积，使用完毕也得不到释放。

在单线程环境中返回指向静态局部变量的指针的函数无法兼容于多线程环境。因为同样

的函数可能在多个线程中同时被调用而导致结果出错。

简而言之，如果共享的变量很多且定义的不好，调试一个多线程的应用程序将会是噩梦

一场。

(2) 共享的文件描述符

在 fork 原语被调用之后，文件描述符自动地被复制，从而子进程得到了一套新的文件描述符。在子进程关闭了某一个从父进程那里继承来的文件描述符之后，此描述符对父进程来说仍然是打开的。

在多线程程序中，很有可能会将同一个文件描述符传递给两个不同的线程。即传递给它们的两个值指向同一个文件描述符。显然如果一个线程中的函数关闭了这个文件，此文件描

述符对此进程中的任何线程来说都已经被关闭。然而其他线程或许仍然需要对此文件描述符的连接。

(3) fork、exec、exit、Signals

所有的线程都共享同一个进程。如果一个线程调用了 exec，系统内核用一个新的程序取代当前的程序从而所有正在运行的线程都会消失。并且如果一个线程执行了 exit，那么整个进程都将结束运行。想想要是线程的运行导致了内存段异常或者系统错误或是线程崩溃，瘫痪的是整个进程，而不是某个线程本身了。

fork 创建了一个新的进程，并把原调用进程的数据和代码复制给这个新的进程。如果线程中的某函数调用了 fork，那么其他的线程是不是也会被复制给新的进程呢？答案是否定的，只有调用 fork 的线程在新的进程中运行。试想一下如果在 fork 发生的时刻，另一个线程正在修改数据，那结果如何呢？在什么情况下这些数据会被复制到新的进程中呢？

信号量(Signal)的使用要比线程复杂多了。进程可以接收任何种类的信号量。那么哪些线程可以收到信号量？是不是所有线程都可以呢？如果这些信号量是由内存段异常或系统错误引发的又将如何？线程与信号量的细节可参考 Unix 的使用手册。

(4) 动手做一做

这一章介绍了线程的基本知识、主要问题以及多线程程序设计细节。对于这一章的最好的练习就是对于同一个问题设计两种不同的解决方案，一个使用线程，另一个使用进程。再看一看哪一个容易设计、编码以及调试；哪一个运行的快一些；哪一个又更适用与兼容 Unix 的各种版本呢？

14.5　线程间互通消息

再看一下上面的多线程字数统计程序。假设你是一个大城市选举的负责人。城市中小一点的选区很快就完成了统计票数的工作，然而你却要等到所有的数字都出来之后才能宣布这个重要的结果。不过你希望在每个选区票数出来之后立即可以看到结果。

在文件中统计数字就像是选区统计票数一样。有些文件比较大，因此就需要校长的时间来做统计。看一看如果下面的命令被运行后，结果是什么样的：

```
twordcount really-big-file tiny-file
```

原线程使用 pthread_wait 来等候第一个和第二个线程返回。在这个例子当中，统计第

一个文件要比第二个文件的时间长的多。那么在第二个线程完成之后，如何来通知原线程呢？

一个线程是如何与另一个线程互通消息的呢？在一个计数线程完成任务之后，它是如何通知原线程它的结果已经产生了呢？对于进程来说，当子进程终止后，系统调用 wait 返回。是不是对于线程的处理也有类似的机制呢？某个线程是否也可以等待其他线程完成？答案是否定的。线程无法按照这种方法工作。因为对于线程而言并没有父线程、子线程的概念，因此并不存在某一个明显的线程可以去通知。

14.5.1　通知选举中心

某选区完成计票后，将其结果发送给管理中心看一下下面的这个方法，此方法是用来从选区得到选票，然后将其发送给管理中心（也许看起来有些古怪，不过线程确实就是用这种方法来与另外的事件互通消息）。

(1) 在选举中心有一个投递选举报告的邮箱。这个邮箱在某一时刻只能接收一份票数报告。

此邮箱前有一面旗帜，它可以被升起来，但很快就会被恢复至原位。

(2) 选举中心等待这面旗帜升起来。

(3) 某选区负责人将选区统计结果放入邮箱中。

(4) 某选区负责人将邮箱的旗帜升起（发送信号）。

(5) 选区中心看到旗帜升起来了，便执行下列步骤：

- 从邮箱中取出选区的统计报告；
- 处理此统计报告；
- 回到原来的地方继续等待，即循环转至步骤(2)。

上面的策略起初看起来有些古怪，但确实很有意义。发送方将数据存入容器中，然后升起一面旗帜来通知接收方数据已经准备好了。

图 14.8 清楚地展现了选举中心与两个选区之间的关系。每一个选区将自己的报告放入邮箱中，然后通知选举中心来取。最后选举中心处理了报告。在这个例子中升旗帜的技术术语叫做发送信号，即接收方在等待信号的到来。当然，这里只是个比喻，对旗帜的操作与 Unix 里的信号量机制一点关系也没有，两者仅仅是基本思路相同而已。

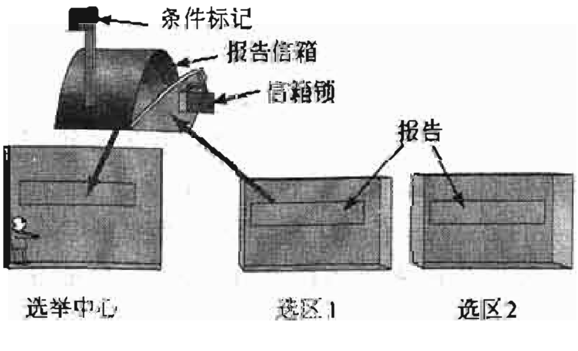


图 14.8　使用加锁的邮箱来传递数据

从图 14.8 中还可以看到另外一样东西：邮箱上有一把锁。邮箱仅仅可以容纳一份报告，因此在某个时刻只能有一人拥有对邮箱访问的权限。虽然锁的加入使邮箱结构看起来有些复杂，但却使得邮箱相当可靠。使用锁机制的邮箱系统，其完整的过程如下。

(1) 选举中心建一个投递选举报告的邮箱。

此邮箱在某个时刻只能容纳一份选举报告。

此邮箱旁有一面旗帜，它可以被升起来，但很快就会被恢复至原位。

此邮箱有一把互斥的锁，可以被锁住或打开。

(2) 选举中心将邮箱锁打开，然后等待旗帜信号的到来。

某选区负责人等在邮箱口，直到它可以将邮箱锁住。

如果邮箱非空，选区负责人先将邮箱锁打开，等待着旗帜信号的到来，然后再将邮箱锁住。

选区负责人将选举报告放入邮箱中。

(3) 选区负责人将旗帜升起，把信号发出去。

选区负责人把邮箱上的锁打开。

(4) 选举中心看到旗帜信号，停止等待。

选举中心锁住邮箱。

选举中心将选举报告拿出。

选举中心处理该选举报告。

选举中心升起旗帜，以防某一选区负责人已经等待了很久。

选举中心转向步骤(2)。

14.5.2　使用条件变量编写程序

下面将计算投票数目的系统转换成字数统计的程序。计票系统使用了三种设备：容器、旗帜和锁。这三种不同的设备对应了线程编程中的三项：一个变量保存数据、一个条件对象和一个互斥量。图 14.9 展示了三个线程和三个变量。一个变量用作指向字数计数器的指针，一个变量用作条件对象，而另一个变量用作互斥量。

研究一下程序的内在逻辑。原线程启动了两个计数线程然后开始等待结果的到来。特别地，原线程调用 pthread_cond_wait 函数来等待“旗帜升起”。这个系统调用将原线程挂起。

当某一计数线程完成计数后，此线程通过把指针存入“邮箱”变量的方法来传递结果。首先，此线程对此邮箱加锁；然后线程检查邮箱；如果邮箱非空，线程把邮箱锁打开并等待着信号的到来；之后，线程再一次把邮箱锁住，并把结果放入邮箱；最后，计数线程调用了函数 pthread_cond_signal，将条件变量 flag 这面“旗帜”升起来。

此时由于执行 pthread_cond_wait 而挂起等待条件变量 flag 变化的原线程被计数线程发出去的信号唤醒了。原线程急切地想冲过去打开邮箱，然而此时的邮箱仍然被计数线程锁在那里。

当计数线程通过调用 pthread_mutex_unlock 把邮箱锁打开之后，原线程终于得到了对这把锁的控制权。原线程将选举报告从邮箱中拿出来，在屏幕上显示出来，再将其加到总数

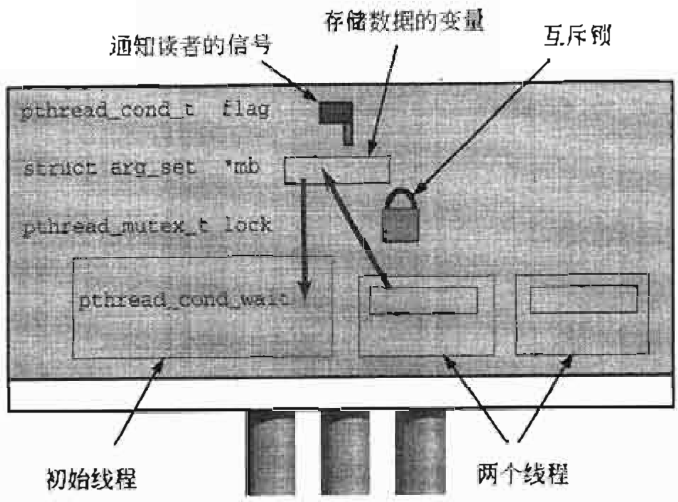


图 14.9　用加锁的变量机制来传递数据

中去。然后原线程发出信号，以防别的计数线程正在等待。最后原线程循环回去，继续调用 pthread_cond_wait 函数，自动将互斥量解锁，并将自己挂起直到下一次的信号到来。

上面一段所讨论的步骤恰好对应于投票系统的原型，也同样对应于下面将看到的 twordcount4.c 中的代码：

```
/* twordcount4.c - threaded word counter for two files.
 *      - Version 4: condition variable allows counter
 *                   functions to report results early
 */

#include <stdio.h>
#include <pthread.h>
#include <ctype.h>

struct arg_set {                          /* two values in one arg  */
        char * fname;                     /* file to examine */
        int count;                        /* number of words */
};
struct arg_set * mailbox;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t flag = PTHREAD_COND_INITIALIZER;

main(int ac, char * av[])
{
    pthread_t t1, t2;                     /* two threads */
    struct arg_set args1, args2;          /* two argsets */
    void     * count_words(void * );
    int      reports_in = 0;
```

```
    int     total_words = 0;

    if ( ac!= 3 ){
        printf("usage: %s file1 file2\n", av#[0];
        exit(1);
    }
    pthread_mutex_lock(&lock);                  /* lock the report box now */

    args1.fname = av[1];
    args1.count = 0;
    pthread_create(&t1, NULL, count_words, (void *) &args1);

    args2.fname = av[2];
    args2.count = 0;
    pthread_create(&t2, NULL, count_words, (void *) &args2);

    while( reports_in < 2 ){
        printf("MAIN: waiting for flag to go up\n");
        pthread_cond_wait(&flag, &lock);        /* wait for notify */
        printf("MAIN: Wow! flag was raised, I have the lock\n");
        printf("%7d: %s\n", mailbox->count, mailbox->fname);
        total_words += mailbox->count;
        if ( mailbox == &args1)
            pthread_join(t1,NULL);
        if ( mailbox == &args2)
            pthread_join(t2,NULL);
        mailbox = NULL;
        pthread_cond_signal(&flag);
        reports_in++;
    }
    printf("%7d: total words\n", total_words);
}
void *count_words(void *a)
{
    struct arg_set *args = a;               /* cast arg back to correct type */
    FILE *fp;
    int c, prevc = '\0';

    if ( (fp = fopen(args->fname, "r"))!= NULL ){
        while( ( c = getc(fp))!= EOF ){
            if ( ! isalnum(c) && isalnum(prevc) )
                args->count++;
            prevc = c;
```

```
        }
        fclose(fp);
    } else
        perror(args->fname);
    printf("COUNT: waiting to get lock\n");
    pthread_mutex_lock(&lock);                  /* get the mailbox */
    printf("COUNT: have lock, storing data\n");
    if ( mailbox != NULL )
        pthread_cond_wait(&flag,&lock);
    mailbox = args;                             /* put ptr to our args there */
    printf("COUNT: raising flag\n");
    pthread_cond_signal(&flag);                 /* raise the flag */
    printf("COUNT: unlocking box\n");
    pthread_mutex_unlock(&lock);                /* release the mailbox */
    return NULL;
}
```

下面的运行显示了时间的发生顺序：

```
$ cc twordcount4.c -lpthread -o twc4
$ ./twc4 /etc/group /usr/dict/words
COUNT: waiting to get lock
MAIN: waiting for flag to go up
COUNT: have lock, storing data
COUNT: raising flag
COUNT: unlocking box
MAIN: Wow! Flag was raised, I have the lock
      195: /etc/group
MAIN: waiting for flag to go up
COUNT: waiting to get lock
COUNT: have lock, storing data
COUNT: raising flag
COUNT: unlocking box
MAIN: Wow! flag was raised, I have the lock
       45419: /usr/dict/words
  45614: total words
$
```

14.5.3 使用条件变量的函数

在邮箱上用来通知其他线程的旗帜就是一个条件变量。下面函数的作用就是使用条件变量进行通信。

pthread_cond_wait		
目标	使线程挂起，等待某条件变量的信号	
头文件	#include <pthread.h>	
函数原型	int pthread_cond_wait (pthread_cond_t * cond, pthread_mutex_t * mutex);	
参数	cond	指向某条件变量的指针
	mutex	指向互斥锁对象的指针
返回值	0	成功返回
	errcode	错误

pthread_cond_wait 使线程挂起直到另一个线程通过条件变量发出消息。pthread_cond_wait 函数总是和互斥锁在一起使用。此函数先自动释放指定的锁，然后等待条件变量的变化。如果在调用此函数之前，互斥量 mutex 并没有被锁住，函数执行的结果是不确定的。在返回原调用函数之前，此函数自动将指定的互斥量重新锁住。

pthread_cond_signal		
目标	唤醒一个正在等候的线程	
头文件	#include <pthread.h>	
函数原型	int pthread_cond_signal(pthread_cond_t * cond);	
参数	cond	指向某条件变量的指针
返回值	0	成功返回
	errcode	错误

pthread_cond_signal 函数通过条件变量 cond 发消息。若没有线程等候消息，什么都不会发生；若是多个线程都在等待，只唤醒它们中的一个。

14.5.4　回到 Web 服务器例子

通过前面的学习，大家已经了解了 POSIX 线程系统最基本知识和技巧，包括如何创建新的线程、如何等候线程返回、如何安全地在线程间共享数据以及线程如何与其他线程互通消息。相信大家已经有足够多的知识可以完成 Web 服务器与复杂动画的制作。

14.6　多线程的 Web 服务器

前面的章节已经写了一个 Web 服务器的程序。服务器使用 fork 系统调用创建新的进程来处理客户端的请求。Web 服务器需要完成三种操作：将目录列表返回；将文件内容返回；以及将 CGI 程序的输出返回。

服务器需要新的进程来运行 CGI 程序，但是并不需要新的进程来读取目录列表和文件内容。

14.6.1　Web 服务器程序的改进

这里对前一个版本的 Web 服务器程序作了很多修改。最重要的是使用函数 pthread_create 替换了 fork。在新的版本中客户端的请求不再由单独的进程来处理，而是由同一进程的多个线程来处理。

另外还做了两个改动：首先，移除了 CGI 的功能。后面的章节会把这个功能加上去。其次，自己写了一个对目录进行列表的函数，而以前的版本是通过调用 exec 执行标准 ls 命令来完成对目录的列表的。

14.6.2　在多线程的版本允许一个新的功能

多线程的特性允许我们添加一个新的功能：内部统计。服务器的运行者通常希望知道服务器的运行时间、接收的客户端请求的数目以及发送回客户端的数据量。

因为对于所有的请求共享内存空间，可以使用共享变量的方式来进行统计。那么用户如何访问这些统计数据呢？这里加入一个特殊的 URL：status。当远程用户请求此 URL 时，服务器将内部的统计数据发给客户端。

14.6.3　防止僵尸线程(Zombie Threads)：独立线程

现在来考虑另一个技术细节。本章中所提到所有的程序中都使用了 pthread_join 函数来等待线程返回。每个线程都占用了系统资源。如果程序员忘记使用 pthread_join 来收回线程，这些被线程所占用的资源就无法被回收，类似于用 malloc 来分配的空间却没有用 free 释放掉一样。

在字数统计的程序中，原线程不得不等待所有的计数线程返回之后，才可以收集数据。然而 Web 服务器却没有理由等待处理请求的线程返回。因为原线程不需要从这些线程得到任何返回数据。

这里同样可以创建不需要返回的线程，称之为独立线程(Detached Threads)。当函数执行完毕之后，独立线程自动释放它所占用的所有的资源，它们自身甚至也不允许等待其他的线程返回。可以通过传递一个特殊的属性参数给函数 pthread_create 来创建一个独立线程。

```
/* creating a detached thread */
pthread_t          t;
pthread_attr_t attr_detached;
pthread_attr_init(&attr_detached);
pthread_attr_setdetached(&attr_detached,PTHREAD_CREATE_DETACHED);
pthread_create(&t,&attr_detached,func,arg);
```

14.6.4　Web 服务器代码

采用多线程方法实现 Web 服务器的完整代码如下：

```
/* twebserv.c - a threaded minimal web server (version 0.2)
```

```
 *   usage: tws portnumber
 * features: supports the GET command only
 *          runs in the current directory
 *          creates a thread to handle each request
 *          supports a special status URL to report internal state
 * building: cc twebserv.c socklib.c -lpthread -o twebserv
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#include <dirent.h>
#include <time.h>

/* server facts here */

time_t   server_started ;
int      server_bytes_sent;
int      server_requests;

main(int ac, char *av[])
{
    int        sock, fd;
    int        *fdptr;
    pthread_t worker;
    pthread_attr_t attr;

    void *handle_call(void *);

    if ( ac == 1 ){
        fprintf(stderr,"usage: tws portnum\n");
        exit(1);
    }
    sock = make_server_socket( atoi(av[1]) );
    if ( sock == -1 ) { perror("making socket"); exit(2); }

    setup(&attr);

    /* main loop here: take call, handle call in new thread */

    while(1){
        fd = accept( sock, NULL, NULL );
        server_requests++;
```

```
        fdptr = malloc(sizeof(int));
        *fdptr = fd;
        pthread_create(&worker,&attr,handle_call,fdptr);
    }
}

/*
 * initialize the status variables and
 * set the thread attribute to detached
 */
setup(pthread_attr_t *attrp)
{
    pthread_attr_init(attrp);
    pthread_attr_setdetachstate(attrp,PTHREAD_CREATE_DETACHED);

    time(&server_started);
    server_requests = 0;
    server_bytes_sent = 0;
}

void *handle_call(void *fdptr)
{
    FILE   *fpin;
    char   request[BUFSIZ];
    int    fd ;

    fd = *(int *)fdptr;
    free(fdptr);                    /* get fd from arg */

    fpin = fdopen(fd, "r");         /* buffer input */
    fgets(request,BUFSIZ,fpin);     /* read client request */
    printf("got a call on %d: request = %s", fd, request);
    skip_rest_of_header(fpin);

    process_rq(request, fd);        /* process client rq */

    fclose(fpin);
}

/* ---------------------------------------------------------------- *
   skip_rest_of_header(FILE *)
   skip over all request info until a CRNL is seen
   ---------------------------------------------------------------- */

skip_rest_of_header(FILE *fp)
{
    charbuf[BUFSIZ] = [];
    while( fgets(buf,BUFSIZ,fp) != NULL && strcmp(buf,"\r\n,")
```

```
        != 0 )
        ;
}

/* ------------------------------------------------------------------------ *
   process_rq( char *rq, int fd )
   do what the request asks for and write reply to fd
   handles request in a new process
   rq is HTTP command: GET /foo/bar.html HTTP/1.0
   ------------------------------------------------------------------------ */

process_rq( char *rq, int fd)
{
    char  cmd[BUFSIZ], arg[BUFSIZ];

    if ( sscanf(rq, ,"%s%s,", cmd, arg) != 2 )
        return;
    sanitize(arg);
    printf("sanitized version is %s\n", arg);

    if ( strcmp(cmd,"GET") != 0 )
        not_implemented();
    else if ( built_in(arg, fd) )
        ;
    else if ( not_exist( arg ) )
        do_404(arg, fd);
    else if ( isadir( arg ) )
        do_ls( arg, fd );
    else
        do_cat( arg, fd );
}
/*
 * make sure all paths are below the current directory
 */
sanitize(char *str)
{
    char *src, *dest;

    src = dest = str;

    while( *src ){
        if ( strncmp(src,"/../",4) == 0 )
            src += 3;
        else if ( strncmp(src,"//",2) == 0 )
            src++;
        else
```

```
            *dest++ = *src++;
        }
        *dest = '\0';
        if ( *str == '/' )
            strcpy(str,str+1);

        if ( str[0] == '\0' || strcmp(str,"./") == 0
                || strcmp(str,"./..") == 0 )
            strcpy(str,".");
}

/* handle built - in URLs here. Only one so far is "status" */
built_in(char *arg, int fd)
{
        FILE *fp;

        if ( strcmp(arg,"status") != 0 )
            return 0;
        http_reply(fd, &fp, 200, "OK", "text/plain",NULL);

        fprintf(fp,"Server started: %s", ctime(&server_started));
        fprintf(fp,"Total requests: %d\n", server_requests);
        fprintf(fp,"Bytes sent out: %d\n", server_bytes_sent);
        fclose(fp);
        return 1;
}

http_reply(int fd, FILE **fpp, int code, char *msg, char *type, char *content)
{
        FILE    *fp = fdopen(fd, "w");
        int     bytes = 0;

        if ( fp != NULL ){
            bytes = fprintf(fp,"HTTP/1.0 %d %s\r\n", code, msg);
            bytes += fprintf(fp,"Content-type: %s\r\n\r\n", type);
            if ( content )
                bytes += fprintf(fp,"%s\r\n", content);
        }
        fflush(fp);
        if ( fpp )
            *fpp = fp;
        else
            fclose(fp);
        return bytes;
}
```

```
/* ------------------------------------------------------------------ *
   simple functions first:
      not_implemented(fd)   unimplemented HTTP command
      and do_404(item,fd)   no such object
   ------------------------------------------------------------------ */

not_implemented(int fd)
{
    http_reply(fd,NULL,501,"Not Implemented","text/plain"
         "That command is not implemented");
}

do_404(char * item, int fd)
{
    http_reply(fd,NULL,404,"Not Found","text/plain",
         "The item you seek is not here");
}

/* ------------------------------------------------------------------ *
   the directory listing section
   isadir() uses stat, not_exist() uses stat
   ------------------------------------------------------------------ */

isadir(char * f)
{
    struct stat info;
    return ( stat(f, &info) != - 1 && S_ISDIR(info.st_mode) );
}

    not_exist(char * f)
{
    struct stat info;
    return( stat(f,&info) == - 1 );
}

do_ls(char * dir, int fd)
{
    DIR      * dirptr;
    struct  dirent * direntp;
    FILE     * fp;
    int      bytes = 0;

    bytes = http_reply(fd,&fp,200,"OK","text/plain",NULL);
    bytes += fprintf(fp,"Listing of Directory % s\n", dir);

    if ( (dirptr = opendir(dir)) != NULL ){
        while( direntp = readdir(dirptr) ){
```

```
        bytes += fprintf(fp, "%s\n", direntp->d_name);
      }
      closedir(dirptr);
    }
    fclose(fp);
    server_bytes_sent += bytes;
}

/* ------------------------------------------------------------------------ *
   functions to cat files here.
   file_type(filename) returns the 'extension': cat uses it
   ------------------------------------------------------------------------ */

char * file_type(char * f)
{
    char * cp;
    if ( (cp = strrchr(f,'.' )) != NULL )
      return cp + 1;
    return "   -";
}

/* do_cat(filename,fd): sends header then the contents */

do_cat(char * f, int fd)
{
    char * extension = file_type(f);
    char * type = "text/plain ";
    FILE * fpsock, * fpfile;
    intc;
    intbytes = 0;

    if ( strcmp(extension, "html ") == 0 )
      type = "text/html";
    else if ( strcmp(extension, "gif") == 0 )
      type = "image/gif";
    else if ( strcmp(extension, "jpg") == 0 )
      type = "image/jpeg";
    else if ( strcmp(extension, "jpeg") == 0 )
      type = "image/jpeg";

    fpsock = fdopen(fd, "w");
    fpfile = fopen( f , "r");
    if ( fpsock != NULL && fpfile != NULL )
    {
      bytes = http_reply(fd,&fpsock,200,"OK",type,NULL);
      while( (c = getc(fpfile) ) != EOF ){
```

```
            putc(c, fpsock);
            bytes++;
        }
        fclose(fpfile);
        fclose(fpsock);
    }
    server_bytes_sent += bytes;
}
```

此程序虽然能够正常工作，但还是有一个问题：统计的功能使用共享变量机制，但共享的变量并未被互斥锁所保护。加入互斥锁的功能就留给大家作为练习。

14.7　线程和动画

Web 服务器程序不需要线程也可以工作，因为可以使用 fork 来处理并发的请求。然而如果没有引入线程机制，Web 浏览器却无法轻易地使画面和广告动起来。对线程的下一个应用是如何用多线程来控制动画。

在视频游戏那一章中，使用了定时器来控制动画。定时器以一个特定的时间间隔来发送 SIGALRM 消息，信号处理者使用计数器来决定何时移动图片。

14.7.1　使用线程的优点

信号处理者和定时器的机制虽然可以完成工作，但线程机制更好地匹配了内部和外部的结构。在外部，用户可以看见两个独立的活动流程：动画和键盘控制，如图 14.10 所示。



图 14.10　动画图片及其键盘控制

在内部，线程可以将控制动画代码和键盘输入代码分开。如图 14.11 所示，线程间通过共享变量方式定义位置、动画速度。

当然，画面的移动还是通过隐藏的定时器来完成的，但多线程的解决方案让我们更关注于程序的组织结构。

多线程与原先的方法相比还有另外一个好处。现代的线程库允许不同的线程运行在不同的处理器芯片上，从而实现了真正意义上的并行。对于动画而言，其轨道、旋转以及纹理的绘制都需要复杂的计算，因此在多个处理器上运行线程可以提供更快的处理速度和更加精细的画面。

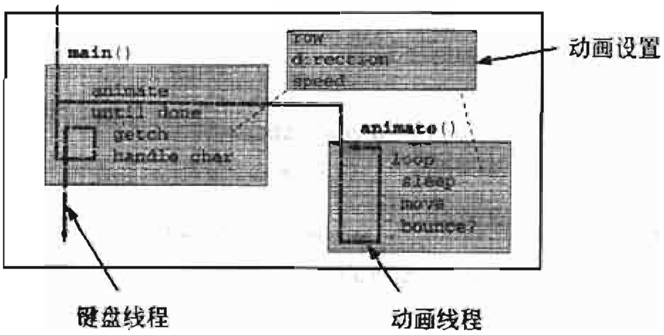


图 14.11　动画线程及键盘线程

14.7.2　多线程版本的 bounce1d.c

比较一下原先版本 bounce1d.c 与新的两线程版本 tbounce1d.c 的区别：

```
/* h tbounce1d.c: controlled animation using two threads
 *      note    one thread handles animation
 *       other thread handles keyboard input
 * compile cc tbounce1d.c -lcurses -lpthread -o tbounce1d
 */

#include <stdio.h>
#include <curses.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

/* shared variables both threads use. These need a mutex. */

#define MESSAGE " hello"

int row;                        /* current row */
int col;                        /* current column  */
int dir;                        /* where we are going */
int delay;                      /* delay between moves */

main()
{
    int ndelay;                 /* new delay */
    int c;                      /* user input */
    pthread_t msg_thread;       /* a thread */
    void * moving_msg();

    initscr();                  /* init curses and tty */
    crmode();
    noecho();
```

```
    clear();

    row = 10;               /* start here */
    col = 0;
    dir = 1;                /* add 1 to row number */
    delay = 200;            /* 200ms = 0.2 seconds */

    if ( pthread_create(&msg_thread,NULL,moving_msg,MESSAGE) ){
        fprintf(stderr,"error creating thread");
        endwin();
        exit(0);
    }

    while(1) {
        ndelay = 0;
        c = getch();
        if ( c == 'Q' ) break;
        if ( c == ' ' ) dir = -dir;
        if ( c == 'f' && delay > 2 ) ndelay = delay/2;
        if ( c == 's') ndelay = delay * 2 ;
        if ( ndelay > 0 )
            delay = ndelay ;
    }
    pthread_cancel(msg_thread);
    endwin();
}

void *moving_msg(char *msg)
{
    while( 1 ) {
        usleep(delay*1000);             /* sleep a while */
        move( row, col );               /* set cursor position */
        addstr( msg );                  /* redo message */
        refresh();                      /* and show it */

        /* move to next column and check for bouncing */

        col += dir;                     /* move to new column */

        if ( col <= 0 && dir == -1 )
            dir = 1;
        else if ( col+strlen(msg) >= COLS && dir == 1 )
            dir = -1;

    }
}
```

新版本的动画程序与老版本的单线程程序有何区别呢？最大的不同之处在于 main 函

数中创建了一个新的线程来执行 moving_msg 函数。moving_msg 函数执行了一个简单的循环：sleep，move，检查跳跃，repeat。同时，在同一个进程的另一部分，main 函数也执行一个简单的循环：getch，处理，repeat。

修改后的程序仍然用全局变量表示球的状态。在基于中断的版本中，必须使用全局变量，因为无法将参数传给信号处理者。然而线程机制却允许线程接收参数，因此可以像上面字数统计程序的第三、第四版本一样，通过创建结构体，并将其地址传给线程的方式来改进程序。

14.7.3　基于多线程机制的多重动画：tanimate.c

怎样才可以同时使多条消息活动起来呢？多线程的字数统计程序并发地运行了多个字数统计函数的实例，每一个调用都传递给此函数一个文件名和一个独立的计数器。用同样的道理可以运行并行的动画。

下面的多线程动画程序 tanimate.c 是 tbounce1.c 的一个扩展。tanimate.c 最多可以接受 10 个命令行的参数，使每一个参数在不同的行上移动，并且有自己独立的速度和方向。例如，下面的命令

```
tanimate 'Buy this' 'Drive this car' 'Spend Money here' Consume '1Buy!  '
```

将产生一个包含多种跳动的动画消息，如图 14.12 所示。

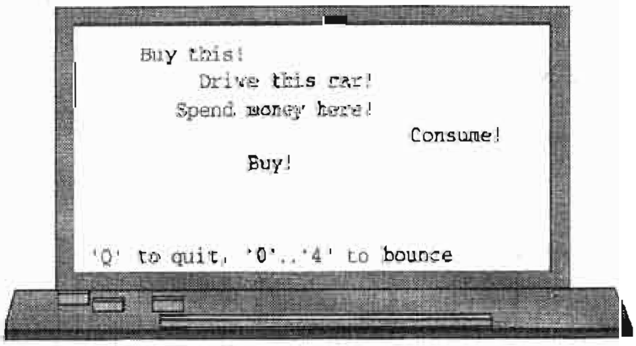


图 14.12　多种跳动的动画消息

用户可以通过按键“0”、“1”等使处在该行上的消息跳动。

也可以使一组字符串活动起来，甚至是那些 Unix 的工具，可以尝试下面的命令：

```
tanimate    'ls'
tanimate    'users'
tanimate    'date'
```

tanimate 在不同的线程中运行控制动画的函数。这个函数的每一个实例都接收到原始线程所传的一组不同的参数。参数指定了消息名称、行号、移动方向以及速度：

```
/* tanimate.c: animate several strings using threads, curses,
 * usleep()
```

```
 *
 * bigidea one thread for each animated string
 *      one thread for keyboard control
 *      shared variables for communication
 * compile      cc tanimate.c -lcurses -lpthread -o tanimate
 * to do      needs locks for shared variables
 *       nice to put screen handling in its own thread
 */

#include <stdio.h>
#include <curses.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#define    MAXMSG     10              /* limit to number of strings */
#define     TUNIT     20000           /* timeunits in microseconds */

struct propset {
        char *str;                    /* the message */
        int row;                      /* the row */
        int delay;                    /* delay in time units */
        int dir;                      /* +1 or -1 */
  };

pthread_mutex_t mx = PTHREAD_MUTEX_INITIALIZER;

int main(int ac, char *av[])
{
   int             c;                 /* user input */
   pthread_t       thrds[AXMSG];      /* the threads */
   struct propset  props[AXMSG];      /* properties of string */
   void            *animate();        /* the function */
   int             num_msg ;          /* number of strings */
   int             i;

   if ( ac == 1 ){
     printf("usage: tanimate string ..\n");
     exit(1);
   }

   num_msg = setup(ac-1,av+1,props);

   /* create all the threads */
   for (i=0 ; i<num_msg; i++)
      if ( pthread_create(&thrds[i], NULL, animate, &props[i])){
         fprintf(stderr,"error creating thread");
         endwin();
```

```
        exit(0);
      }

    /* process user input */
    while(1) {
       c = getch();
       if ( c == 'Q') break;
       if ( c == ' ' )
          for (i = 0;i<num_msg;i++)
              props[i].dir = -props[i].dir;
       if ( c >= '0' && c <= '9' ){
          i = c - '0';
          if ( i < num_msg )
             props[i].dir = -props[i].dir;
       }
    }

    /* cancel all the threads */
    pthread_mutex_lock(&mx);
    for (i=0; i<num msg; i++ )
        pthread_cancel(thrds[i]);
    endwin();
    return 0;
    }

int setup(int nstrings, char *strings[], struct propset props[])
{
    int num_msg = ( nstrings > MAXMSG ? MAXMSG : nstrings );
    int i;

    /* assign rows and velocities to each string */
    srand(getpid());
    for (i = 0 ; i<num_msg; i++ ){
       props[i].str = strings[i];                   /* the message */
       props[i].row = i;                            /* the row */
       props[i].delay = 1+(rand()%15);              /* a speed */
       props[i].dir = ((rand()%2)? 1:-1);           /* +1 or -1 */
    }

    /* set up curses */
    initscr();
    crmode();
    noecho();
    clear();
    mvprintw(LINES-1,0,"'Q' to quit, '0'..'%d' to bounce",
             num_msg-1);
```

```
    return num_msg;
}

/* the code that runs in each thread */
void * animate(void * arg)
{
    struct propset * info = arg;                /* point to info block */
    int len = strlen(info->str) + 2;            /* + 2 for padding */
    int col = rand() % (COLS - len - 3);        /* space for padding */

    while( 1 )
    {
        usleep(info->delay * TUNIT);

        pthread_mutex_lock(&mx);                /* only one thread */
            move( info->row, col );             /* can call curses */
            addch( ' ' );                       /* at the same time */
            addstr( info->str );                /* Since I doubt it is */
            addch( ' ' );                       /* reentrant */
            move(LINES - 1,COLS - 1);           /* park cursor */
            refresh();                          /* and show it */
        pthread_mutex_unlock(&mx);              /* donewith curses */

        /* move item to next column and check for bouncing */

        col += info->dir;

        if ( col <= 0 && info->dir == -1 )
            info->dir = 1;
            else if (col + len >= COLS && info->dir == 1 )
                    info->dir = -1;
        }
    }
```

14.7.4 tanimate.c 中的互斥量

上面的程序，整个代码分为三段：初始化、控制移动消息的函数和一个读取和处理用户输入的循环。用户输入循环在初始线程中运行，而控制动画的函数却是运行在好几个线程中。tanimate 可以最多同时有 11 个线程在运行。在线程并行运行过程中，共享资源是什么？又如何来防止线程间的共享冲突呢？

1. 数据冲突：互斥量的动态初始化

控制动画的函数可以使用或修改作为参数传递给它的结构体成员的值，它们包括位置、速度以及运动方向。当用户使一条消息跳动之后，输入线程修改了结构体中成员 dir 的值。大家都知道，共享变量需要互斥量来防止数据的冲突。那么是否需要为所有的方向变量(即结构体中的成员 dir)增加一个互斥量呢？

一个比较好的方案是在每一个结构体中建一个互斥量。当控制动画的线程和用户输入线程读取或修改结构体中共享变量的时候，这个互斥量开始工作。修改后的结构体定义如下：

```
struct  propset{
                char          *str;          /* the message */
                int           row;           /* the row */
                int           delay;         /* delay in time units */
                int           dir;           /*  +1 or  -1 */
        pthread_mutex_t lock;                /* a mutex for dir */
   }
```

setup 中的初始化程序如下：

```
for (i = 0; i<num_msg; i++){
        props[i].str  =  strings[i];                /* the message */
        props[i].row  =  i;                         /* the row */
        props[i].delay  =  1 + (rand() % 15);       /* a speed */
        props[i].dir  =  ((rand() % 2)? 1; - 1);    /*  +1 or  -1 */
    pthread_mutex_init(&props[i].lock,NULL);
  }
```

程序中其他的改进留给大家作为练习。

2. 屏幕冲突：临界区

方向变量并不是惟一的共享资源。修改屏幕和光标位置的各函数同样也被所有动画线程共享。可使用互斥量 mx 来防止对这些函数的同步访问冲突。

看一下 animate 中的屏幕控制调用：move、addch、addstr 和 refresh。如果两个线程并发地按照这个顺序执行指令，那么结果会怎样？举例来说，如果两个线程交替地调用：move、move、addch、addch、addstr、addstr…，会导致什么样的结果？

第一个线程使用 move 将光标置于某个屏幕位置，然后第二个线程又会将其移到别的地方去。然而这时第一个线程以为光标还在原处，就将一些文字输出到这个位置，结果显然是错误的！

由于设置光标的库函数不会意识到线程的存在，所以对某个特定函数的访问并不会因为多线程的存在而受干扰。但为了保证某一时刻只有一个线程可以访问设置光标的函数，这里同样使用了互斥量机制。

光标控制函数库包含了许多内部的结构体。就像用互斥量来保护自定义数据结构一样，这里也使用互斥量来防止对系统系统库内部的数据结构的访问冲突。

14.7.5　屏幕控制线程

使用互斥量并不是防止屏幕控制冲突的惟一方法。另一个方法就是创建一个新的线程来处理所有对屏幕控制函数的调用，如图 14.13 所示。

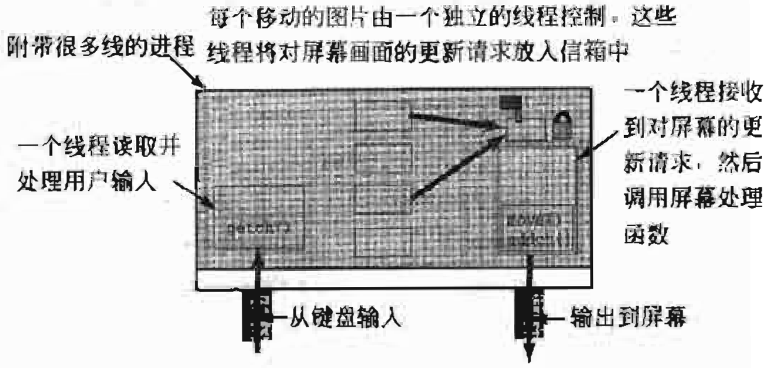


图 14.13　独立处理屏幕控制的线程

可以把这个屏幕控制线程看成在一个大公司中的公共关系部门。任何想要向媒体发送消息的部门都必须向公共关系部门发送请求。公共关系部门里的员工只关心如何将消息发送出去。

其实这个屏幕控制线程在功能上就像这个公关部一样，任何希望向屏幕发消息的线程都必须向这个屏幕管理线程发送请求。

采用这种机制之后，每个线程所发送的请求就应该包含这些信息：行号、列号和消息字符串。控制动画的线程发出消息，屏幕管理线程接到消息后，将其显示在屏幕上。

这种生产者（发消息线程）——消费者（接收消息线程）的设计类似于前面学习的多线程版的字数统计程序：需要一个存储变量来放置消息，通过互斥量来避免对此存储变量的访问冲突，以及一个条件变量，在动画线程发送消息之后，可以通过这个条件变量将屏幕控制线程唤醒。

对于屏幕控制功能的集中化和抽象化设计使得程序更加灵活。就算屏幕显示系统更换了，也只需改变屏幕控制线程中使用的函数。屏幕控制线程甚至还可以发起同远端显示服务器的一个会话，通过管道或套接字将消息发过去，然而这一切对于控制动画的线程来说都是透明的。改进这个程序就留给大家作为练习完成。

小　　结

1. 主要内容

- 执行线路即为程序的控制流程。pthreads 的线程库允许程序在同一时刻运行多个函数。
- 同时执行的各函数都拥有自己的局部变量，但共享所有的全局变量和动态分配的数据空间。
- 当线程共享变量时，必须保证它们不会发生共享冲突。线程使用互斥锁来保证在某一时刻只有一个线程在对共享变量访问。
- 线程间通过条件变量来互相通知和同步数据。一个线程挂起并等待着条件变量按照某种特定方式变化，而另一个线程则发信号使得条件变量发生变化。
- 线程需要使用互斥量来避免对于共享资源操作函数的访问冲突。非重入的函数必须

按照这种方式进行保护。

2. 下一步做什么

一个程序可以包括若干进程，进程之间通过管道、文件、socket 和信号进行通信。程序也可以包含多个线程，它们之间通过共享变量、文件、锁以及信号进行通信。前一章主要学习了进程间的通信。那么 Unix 中提供了多少种通信方式呢？如何为你的应用程序选择最合适的通信方法呢？下一章将给出这些问题的答案。

3. 习题

14.1 线程基本操作的练习。对 hello_multi.c 做如下修改：
首先，多加几条消息到程序中，接着为新加的消息创建线程。
然后修改 print_msg 函数中的循环次数，使其与所要打印的字符串的长度一致。
在每一条 pthread_join 语句之后打印信息，以观察程序的执行情况，并解释结果。

14.2 对 tanimate 使用管道命令，tanimate 就可以从标准输入中读取它的字符串列表。那么如下的命令：

```
who|tanimate
```

就可以起作用了。将这一功能添加进去并不是一件小事。需要首先从标准输入读取字符行，然后将标准输入重定向到终端，这样屏幕控制线程就可以非标准的模式读取字符了(打开 /dev/tty 并将它复制到标准输入)。

14.3 在视频游戏那一章中，大家一定注意到在代码的临界区使用信号标记来防止访问冲突。将信号标记和信号处理机制与线程、互斥锁以及条件变量机制做一个比较。

4. 编程练习

14.4 在 hello_multi.c 中，原始线程创建了两个打印线程。写一个新的程序，在打印“hello”的线程中创建一个新的线程来打印“world\n”。哪一个线程等待打印“world\n”的线程返回？为什么？

14.5 twordcount1.c 用了三个线程：初始线程和两个计数线程，但初始线程完成很少的功能。写一个新程序，让原线程统计第一个文件的字数，然后再创建一个线程统计第二个文件的字数。这种方法是否执行得快一点？哪一种设计更好？

14.6 count_words 函数报错说无法打开指定的文件。尽管这样另一个线程仍然继续运行。这样好吗？修改程序使 count_words 函数在打不开文件的时候调用 exit 退出运行。

14.7 如何扩展 twordcount2.c 和 twordcount3.c 中的方法，让程序可以在命令行上接收多个文件？修改这两个程序使它们可以在命令行上接收任意数目的文件名。哪一个版本的程序容易扩展？哪一个更高效呢？

14.8　进程与线程：

(1) 写一个新版本的字数统计程序，用 fork 为每个文件创建新的进程。需要为子进程设计一个系统，使它们可以将结果传回给父进程使用。不要使用进程退出值来返回这个结果，因为这个值不能超出 255。使用 fork 的一个好处就是可以使用标准的 wc －w 命令来统计每个文件中的字符数。

(2) 写一个单线程的字数统计程序。

(3) 将这三个版本的程序(进程、多线程和单线程)做一下比较，哪一个容易设计、容易编码？哪一个执行效率高？哪一个可移植性好？

14.9　扩展 twordcount4. c 程序，使其在命令行上可以接收多于两个文件名。

14.10　将 twordcount4. c 中的全局变量作为 main 函数的局部变量，然后将指向它们的指针作为结构体的成员。再将结构体地址传给线程。

14.11　在 twebserv. c 中加入互斥锁来保护对统计变量的访问。

14.12　删除 tbounce1d. c 中的所有全局变量。定义一个结构体来包含所有跳动文字的信息。

14.13　tbounce1d. c 程序中的共享状态需要互斥锁机制。如果不加锁，会导致什么样的结果？两个线程冲突会造成什么样的结果？

14.14　单字符串的动画程序包含了对速度的控制。用户可以通信按键“s”和“f”来增加和降低动作间的延时，将速度控制加入多字符串动画程序中。

14.15　tanimate. c 中所有的消息都是水平移动的。修改程序使得一些字符串上下移动，而另一些水平移动。如何来控制冲突？

14.16　修改 tanimate 程序，使用一个单独的线程进行屏幕管理。控制动画的线程必须先将消息发送给屏幕管理线程，由屏幕管理线程对消息进行显示。

14.17　MT finger 服务器需要一个多线程的服务器来提供服务。服务器每次接收一行输入。然后服务器在数据库中查询这个字符串，再将匹配这个字符串的记录返回给客户端。服务器运行包含如下步骤：

(1) 将用户数据库载入内存；

(2) 为每一个请求创建一个独立线程(detached thread)；

(3) 服务器记录每秒钟所匹配的记录的数目；

(4) 对于 STATUS 的特殊请求，服务器将返回统计数据；

(5) 在接收到 SIGHUP 消息之后，服务器刷新其内部数据库。

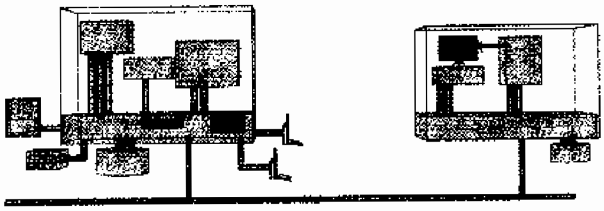
14.18　在 tanimate 那一节的结尾，曾讨论了如何将显示功能与计时和数据处理逻辑分开。写一个客户/服务器版本的 tanimate 程序，用数据报的 socket 将简单的请求信息从 tanimate 程序发送到显示服务器上。

显示服务器支持包含如下两条命令的简单协议：

```
CLEAR                     clears the display
DRAW R C Any string       puts "Any string" at row R,column C
```

修改后的版本不再需要调用屏幕控制函数。它只需向显示服务器发送消息。显示服务器接收这些信息，然后把消息中包含的字符串显示出来。在做这道题目的时候，可以去研究 Unix 使用的 X11 window 系统的设计思想。

第 15 章　进程间通信(IPC)



概念与技巧

- 挂起并等候从多个源端的输入：select 和 poll
- 命名管道
- 共享内存
- 文件锁
- 信号量
- IPC(InterProcess Communication)总览

相关的系统调用与函数

- select、poll
- mkfifo
- shmget、shmat、shmctl、shmdt
- semget、semctl、semop

相关命令

- talk
- lpr

15.1　编程方式的选择

很久以前，当两人试图互相交流的时候，可选择的方式非常的少：交谈或者投掷石块。当代人们的选择多了很多：蜂窝电话、电子邮件、信件、特快专递、自行车投递、网络电话、纸张、显示器上贴的备忘录等，当然也可以直接交谈或投掷石块。每种方法都有其优点和缺点。那么人们如何选择其交流方式呢？

作为一名 Unix 程序员，必须学会如何选择进程间通信的方法。每一种方式都有其优缺点，如何进行选择呢？

本章从学习 talk 开始。人们使用 talk 来互相发送消息，并将会比较讨论各种 Unix 方法，看看它们是如何在进程间传递消息的。

15.2 talk 命令：从多个数据源读取数据

Unix 的 talk 命令是一种通信工具。talk 允许人们在终端间传送信息。talk 甚至可以跨越 Internet 来连接不同机器的终端，如图 15.1 所示。

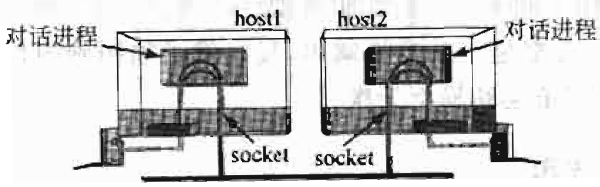


图 15.1 talk 连接网络上的两个终端

使用 talk 命令的时候，屏幕被分为上下两个部分，用户以字符终端模式来操作。输入字符的时候，字符会同时显示在两个窗口中。你所输入的字符会出现在上面的窗口中，而对方输入的字出现在下面的窗口中。talk 使用 socket 进行通信，如图 15.2 所示。

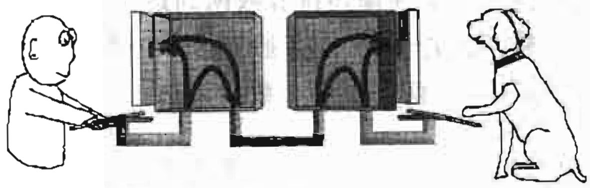


图 15.2 talk 命令的工作方式

talk 命令读取字符并将它们写到目的地，但与所学过的其他程序不同，talk 同时等待从两个文件描述符的输入。

15.2.1 同时从两个文件描述符读取数据

talk 从键盘和 socket 接收数据。从键盘输入读取的字符被复制到屏幕中上半个窗口，并通过 socket 发送出去。同样从 socket 读入的字符被添加到屏幕下面的窗口中。

talk 的用户可以以任意速度和任意顺序输入字符。talk 程序就必须在任何时刻都准备好从任一数据源接收字符，而不像其他的程序可以依靠明确的协议。服务器等待着 read 或 recvfrom 的请求，并用 write 或 sendto 发回一个应答。用户不可能一直在切换自己的角色（输入完之后等待别人的应答，然后再输入……），那么 talk 程序如何解决这个问题呢？talk 当然也不能这样做，如下述代码：

```
while(1){
    read(fd_kbd,&c,1);          /* read from keyboard */
    waddch(topwin,c);           /* add to screen */
    write(fd_sock,&c,1);        /* send to other person */
    read(fd_sock,&c,1);         /* read from other person */
    waddch(botwin,c);           /* add to screen */
}
```

按照上述代码的逻辑，如果对方一直在输入信息，而你却一直在看他发的消息，自己没有输入过，那结果会怎么样呢？程序在第一个 read 调用的时候就挂起了，并不会从你的对方那里读取数据。上面的方法只有在用户不断切换自己角色的时候才可以正常工作。

这里通过调用 fcntl 函数将文件描述符设置为 O_NONBLOCK 标志从而使文件描述符变成非阻塞模式。使用非阻塞模式使得对于 read 的调用立即返回。这个时候，如果并没有字符可以读取，read 调用返回零。虽然非阻塞的方式可以工作，但是它占用了太多的处理器时间。由于每次调用 read 都是一个系统调用，程序就必须回到内核模式工作。这样在等待一个字符到来之前系统可能会切换上千次。

15.2.2　select 系统调用

Unix 系统提供了系统调用 select，它允许程序挂起，并等待从不止一个文件描述符的输入。它的原理很简单：

(1) 获得所需要的文件描述符列表；

(2) 将此列表传给 select；

(3) select 挂起直到任何一个文件描述符有数据到达；

(4) select 设置一个变量中的若干位，用来通知你哪一个文件描述符已经有输入的数据。

下面的程序 selectdemo.c 等待两个设备上数据到达：

```
/* selectdemo.c : watch for input on two devices AND timeout
 *         usage: selectdemo dev1 dev2 timeout
 *        action: reports on input from each file, and
 *                reports timeouts
 */

#include  <stdio.h>
#include  <sys/time.h>
#include  <sys/types.h>
#include  <unistd.h>
#include  <fcntl.h>

#define oops(m,x) { perror(m); exit(x); }

main(int ac, char * av[])
{
    int     fd1, fd2;           /* the fds to watch */
    struct  timeval timeout;    /* how long to wait */
    fd_set  readfds;            /* watch these for input */
    int     maxfd;              /* max fd plus 1 */
    int     retval;             /* return from select */

    if ( ac != 4 ){
      fprintf(stderr,"usage: %s file file timeout", * av);
```

```
        exit(1);
    }

    /** open files **/
    if ( (fd1 = open(av[1],O_RDONLY)) == -1 )
        oops(av[1], 2);
    if ( (fd2 = open(av[2],O_RDONLY)) == -1 )
        oops(av[2], 3);
    maxfd = 1 + (fd1>fd2? fd1:fd2);

    while(1) {
        /** make a list of file descriptors to watch **/
        FD_ZERO(&readfds);              /* clear all bits */
        FD_SET(fd1, &readfds);          /* set bit for fd1 */
        FD_SET(fd2, &readfds);          /* set bit for fd2 */

        /** set timeout value **/
        timeout.tv_sec = atoi(av[3]);   /* set seconds */
        timeout.tv_usec = 0;            /* no useconds */

        /** wait for input **/
        retval = select(maxfd,&readfds,NULL,NULL,&timeout);
        if ( retval == -1 )
            oops("select",4);
        if ( retval > 0 ){
            /** check bits for each fd **/
            if ( FD_ISSET(fd1, &readfds) )
                showdata(av[1], fd1);
            if ( FD_ISSET(fd2, &readfds) )
                showdata(av[2], fd2);
        }
        else
            printf("no input after %d seconds\n", atoi(av[3]));
    }
}
showdata(char *fname, int fd)
{
    char buf[BUFSIZ];
    int n;

    printf("%s: ", fname, n);
    fflush(stdout);
    n = read(fd, buf, BUFSIZ);
    if ( n == -1 )
```

```
        oops(fname,5);
    write(1, buf, n);
    write(1, "\n", 1);
}
```

上面的代码遵循了前面所列出的四步。文件描述符列表以二进制位方式存储在 fd_set 类型的变量中。宏 FD_ZERO、FD_SET 和 FD_ISSET 先将 fd_set 中所有位清除，然后为某文件描述符设置一位，再对该位进行监听。希望同时对两个文件描述符监听，因此调用了两次 FD_SET。

select 调用同时也接受对于超时的处理。如果在指定的时间内，没有数据到达，select 将直接返回。在 selectdemo.c 中，在命令行接收一个字符串作为超时的时间值。用下面的代码对程序进行测试：

```
$ cc selectdemo.c -o selectdemo
$ ./selectdemo /dev/tty /dev/mouse 10
hello
/dev/tty: hello

no input after 4 seconds
no input after 4 seconds
testing
/dev/tty: testing
我移动了鼠标
/dev/mouse: (
/dev/mouse:
/dev/mouse: y
```

这个例子展示了程序如何等待键盘或鼠标输入的到来。当然大家可以设计更加有意思的程序，不只是将输入打印出来，并且对输入进行特定的处理。

select 调用小结如下。

	select
目标	同步的 I/O 复用
头文件	#include <sys/time.h>
函数原型	int = select (int numfds, fd_set * read_set, fd_set * write_set, fd_set * error_set, struct timeval * timeout); void FD_ZERO(fd_set * fdset) void FD_SET(int fd, fd_set * fdset) void FD_CLR(int fd, fd_set * fdset) void FD_ISSET(int fd, fd_set * fdset)

续表

select		
参数	numfds	需要监听的最大 fd 值加 1
	read_set	等待从此集合包括的文件描述符到来的数据
	write_set	等待向这些文件描述符写数据的许可
	error_set	等待这些文件描述符操作的异常
	timeout	超过此时间后函数返回
返回值	−1	发生错误
	0	超时
	num	满足需求的文件描述符的数目

select 同时监视多个文件描述符。在指定情况发生的时候,函数返回。详细一点说,select 监听在三组文件描述符上发生的事件:检查第一组是否可以读取,检查第二组是否可以写入,检查第三组是否有异常发生。每一组的文件描述符被记录到一个二进制位的数组中。这里的 numfds 恰好等于需要监听的最大的文件描述符加 1。

当参数指定的条件被满足或超时的时候,select 函数返回。若指定的条件被满足,select 返回满足条件的文件描述符的数目。

若任一参数为 null,select 将忽略此参数。

15.2.3 select 与 talk

本章并不打算把 talk 的程序写出来,因为关于定位其他的用户和建立连接还有很多步骤要做,比如说:定位对方用户就需要搜寻 utmp 文件。大家已经学习过实现其他所有步骤所需的知识和技巧了。其他还有哪些步骤呢?它们需要哪些系统调用?这两个问题就留给大家回答了。

15.2.4 select 与 poll

也可以使用 poll 调用来代替 select 的功能。select 是由 Berkeley 研制出来的,而 poll 则是贝尔实验室的成果。这两者完成类似的功能,而现代的大部分的 Unix 版本对于两者都支持。

15.3 通信的选择

talk 命令是 Unix 系统程序的一个很好的例子,它很好地体现了进程间如何进行通信和分工合作。talk 中的两个进程读写消息,就好像消息是被存储在常规的磁盘上一样。

talk 中的文件描述符分别对应了键盘、屏幕和 socket(如图 15.3 所示),但它们仍然可以被连接到其他进程或其他设备上去。talk 程序中的进程间数据传输与进程间的操作都是极为重要的部分。要知道选择一个好的通信方式和选择正确的算法或数据结构一样的重要。

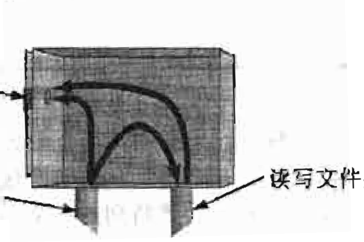


图 15.3　三个文件描述符

15.3.1　一个问题的三种解决方案

问题：从服务器得到数据，将其传给客户端，如何来决定选择哪一种通信方法呢？想一想前面使用流 socket 写过的时间/日期服务器。某一进程知道当前的时间，而另一进程想获取时间信息(如图 15.4 所示)，如何让一个进程从另一个进程得到数据呢？

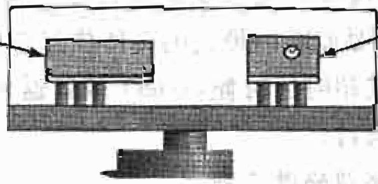


图 15.4　某进程拥有另一进程所要获得的信息

三种解决方案：文件、管道、共享内存。图 15.5 展示了三种不同的方法：一种是已经学习过的，但另外的两种方法却是新的。大家所熟悉的方案是使用文件，而另外的两种新方案是命名管道(named pipe)及共享内存段(share memory segment)策略。这些方法分别通过磁盘、内核以及用户空间进行数据的传输。那么每种方法具体如何应用？各有何优缺点呢？

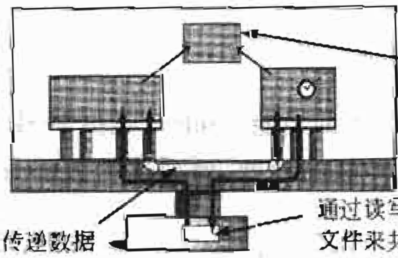


图 15.5　三种传输数据的方法

15.3.2　通过文件的进程间通信

进程间可以通过文件来进行通信。某进程将数据写入文件，别的进程再将数据从文件中读出。

(1) 使用文件进行通信的时间/日期服务器

这里不必写一个完整的 C 程序，下面的一个 shell 脚本就可以完成任务了。

```
#! /bin/sh
# time server - file version
```

```
    while true ; do
    date > /temp/current_date
    sleep 1
done
```

此服务器每隔 1 秒钟将当前的时间和日期写入文件中。输出重定向符“>”每次将该文件内容删除然后重写。

(2) 使用文件进行通信的时间/日期客户器

客户端读取文件的内容：

```
#! /bin/sh
# time client - file version
  cat /tmp/current_date
```

(3) 使用文件的 IPC 小结

访问控制：客户端必须能够读取文件。通过使用标准文件访问权限，可以给予服务器写权限并且限制客户端只有读权限。

多客户端：任意数目的客户可以同时从文件中读取数据。Unix 并不限制同时打开同一文件的进程数目。

竞态条件：服务器通过清空内容再重写的方法来更新文件。如果某客户恰好在清空和重写之间读取文件，那么它得到的将是一个空的或只有部分的内容。

避免竞态条件：服务器和客户端可以使用某种类型的互斥量来避免竞态条件。后面的章节中大家将会学到文件锁的方法。另外，如果服务器在程序中使用 lseek 和 write 函数来替换 create，这样文件永远都不可能为空，因为 write 是一个原子操作，它不会在执行中被打断。

15.3.3 命名管道

通常的管道只能连接相关的进程。常规管道由进程创建，并由最后一个进程关闭。

使用命名管道可以连接不相关的进程，并且可以独立于进程存在(如图 15.6 所示)。称这样的命名管道为 FIFO(先进先出队列)。FIFO 类似于放在草坪上的一段给花园浇水的水管。任何人都可以将此水管的一端放在自己的耳朵边，而另一个人通过水管向对方说话。人们可以通过此水管进行交流，而在没有人使用的时候，水管仍然是存在的。FIFO 可以看作由文件名标志的一根水管。

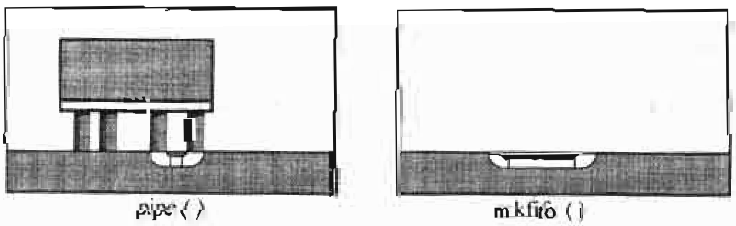


图 15.6　FIFO 与进程独立

1. 使用 FIFO

(1) 如何创建 FIFO?

库函数 mkfifo(char * name, mode_t mode)使用指定的权限模式来创建 FIFO。mkfifo 命令通常调用这个函数。

(2) 如何删除 FIFO?

类似于删除文件,unlink(fifoname)函数可以用来删除 FIFO。

(3) 如何监听 FIFO 的连接?

使用 open(fifoname,O_RDONLY)函数。open 函数阻塞进程直到某一进程打开 FIFO 进行写操作。

(4) 如何通过 FIFO 开始会话?

使用 open(fifoname,O_WRONLY)函数。此时 open 函数阻塞进程直到某一进程打开 FIFO 进行读取操作。

(5) 两进程如何通过 FIFO 进行通信?

发送进程用 write 调用,而监听进程使用 read 调用。写进程调用 close 来通知读进程通信结束。

下面两个 shell 脚本是基于 FIFO 的时间/日期服务的服务器和客户端程序:

```
#! /bin/sh
# time server
    while true ; do
       rm -f /tmp/time_fifo
       mkfifo /tmp/time_fifo
       date > /tmp/time_fifo
    done

#! /bin/sh
# time client
  cat /tmp/time_fifo
```

2. FIFO 类型的 IPC 小结

访问: FIFO 使用与通常文件相同的文件访问。服务器有写权限,而客户端只限于读权限。

多个客户端: 命名管道是一个队列而不是常规文件。写者将字节写入队列,而读者从队列头部移出字节。每个客户端都会将时间/日期的数据移出队列,因此服务器必须重写数据。

竞态条件: FIFO 版本的时间/日期服务器程序完全不存在竞态条件问题。在信息的长度不超过管道的容量的情况下,read 和 write 系统调用只是原子操作。读取操作将管道清空而写入操作又将管道塞满。在读者和写者连通之前,系统内核将进程挂起。因此锁机制在这里并不需要。

时间/日期服务器将数据写入 FIFO 后,将自己挂起直到客户端打开 FIFO 来读取数据。在某些应用程序中,服务器从 FIFO 中读取数据,然后等待客户端把数据写入。大家想想看有没有服务器等待客户端输入的例子?

15.3.4　共享内存

字节流是如何通过文件或 FIFO 来传输的？write 将数据从内存复制到内核缓存中。read 将数据从内核缓存复制到内存中。

如果进程运行在用户空间的不同部分，进程间是如何将数据从内核缓存中复制进复制出的呢？同一个系统里的两个进程通过使用共享的内存段来交换数据。共享的内存段是用户内存的一部分。每一个进程都有一个指向此内存段的指针(如图 15.7 所示)。依靠访问权限的设置，所有进程都可以读取这一块空间中的数据。因此进程间的资源是共享的，而不是被复制来复制去的。共享内存段对于进程而言，就类似于共享变量对于线程一样。

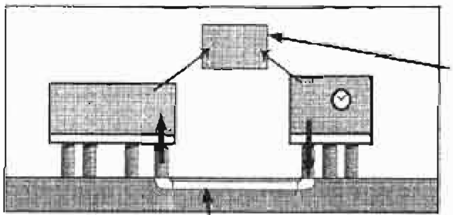


图 15.7　两个进程共享一块内存区域

1. 共享内存段的一些基本概念

- 共享内存段在内存中不依赖于进程的存在而存在。
- 共享内存段有自己的名字，称为关键字(key)。
- 关键字是一个整型数。
- 共享内存段有自己的拥有者以及权限位。
- 进程可以连接到某共享内存段，并且获得指向此段的指针。

2. 使用共享内存段

(1) 如何得到共享内存段？

int seg_id == shmget(key, size-of-segment, flags)

如果内存段存在，函数 shmget 找到它的位置。如果不存在，可以通过在 flags 值中指定一个创建此段和初始化权限模式的请求。

(2) 如何将进程连接到某个共享内存段？

void ptr = *shmat(seg_id,NULL,flags)

shmat 在进程的地址空间中创建共享内存段的部分，并返回一个指向此段的指针。flags 参数用来指定此内存段是否为只读。

(3) 如何与共享内存段进行读写交互？

strcpy(ptr,"hello");

memcpy()、ptr[i]及其他一些通用的指针操作。

3. 使用共享内存段的时间/日期服务器

```
/* shm_ts.c : the time server using shared memory, a bizarre application */
```

```
#include  <stdio.h>
#include  <sys/shm.h>
#include  <time.h>

#define  TIME_MEM_KEY 99             /* like a filename */
#define  SEG_SIZE ((size_t)100)      /* size of segment */
#define  oops(m,x) { perror(m); exit(x); }

main()
{
    int    seg_id;
    char   *mem_ptr, *ctime();
    long   now;
    int    n;

    /* create a shared memory segment */

    seg_id = shmget( TIME_MEM_KEY, SEG_SIZE, IPC_CREAT|0777 );
    if ( seg_id ==    1 )
        oops("shmget", 1);

    /* attach to it and get a pointer to where it attaches */

    mem_ptr = shmat( seg_id, NULL, 0 );
    if ( mem_ptr == ( void * ) -1 )
        oops("shmat", 2);

    /* run for a minute */
    for (n=0; n<60; n++ ){
        time( &now );                      /* get the time */
        strcpy(mem_ptr, ctime(&now));      /* write to mem */
        sleep(1);                          /* wait a sec */
    }

    /* now remove it */
    shmctl( seg_id, IPC_RMID, NULL );
}
```

4. 使用共享内存段的时间/日期客户端

```
/* shm_tc.c : the time client using shared memory, a bizarre application */

#include  <stdio.h>
#include  <sys/shm.h>
#include  <time.h>

#define  TIME_MEM_KEY 99             /* kind of like a port number */
#define  SEG_SIZE ((size_t)100)      /* size of segment */
#define  oops(m,x) { perror(m); exit(x); }
```

```
main()
{
    int   seg_id;
    char  *mem_ptr, *ctime();
    long  now;

    /* create a shared memory segment */

    seg_id = shmget( TIME_MEM_KEY, SEG_SIZE, 0777 );
    if ( seg_id ==    1 )
        oops("shmget",1);

    /* attach to it and get a pointer to where it attaches */

    mem_ptr = shmat( seg_id, NULL, 0 );
    if ( mem_ptr == ( void * ) -1 )
        oops("shmat",2);

    printf("The time, direct from memory: .. %s", mem_ptr);

    shmdt( mem_ptr );                    /* detach, but not needed here */
}
```

5. 共享内存段类型的 IPC 小结

访问：客户端必须有对共享内存段的读权限。共享内存段拥有一个权限系统，它的工作原理和文件权限系统类似。共享内存段有自己的拥有者并且为用户、组或其他成员设置了权限位，来控制他们各自的访问权限。正因为有如此特性，才可以让服务器只有写权限而客户端只有读权限。

多个客户：任意数目的客户都可以同时从共享内存段读数据。

竞态条件：服务器通过调用一个运行在用户空间的库函数 strcpy 来更新共享的内存段。如果客户端正好在服务器向内存段中写入新数据的时候来访问内存段，那么它可能既读到新数据也读到老数据。

避免竞态条件：服务器和客户段必须使用相同的系统来对资源加锁。内核提供了一种进程间加锁的机制，称为信号量机制。在下一节中将会学习这种机制。

15.3.5　各种进程间通信方法的比较

最初的问题是如何使一个进程从另一个进程得到字符串。前面提到的三个方法都可以达到要求。客户端从服务器端得到它们想要的数据。前面已经介绍了四个版本的客户/服务器系统，甚至还可以写出使用数据报或 Unix 域地址的新版本。不过如何决定到底用哪一种方法来实现呢？有什么选择标准吗？

(1) 速度

通过文件或命名管道来传输数据需要更多的操作。系统内核将数据复制到内核空间中，然后再切换回用户空间。对于利用文件进行传输来说，内核将数据复制到磁盘上，然后将数据再从磁盘上复制出去。实际上，在存储器中存储数据比想象中要复杂的多。虚拟内

存系统允许用户空间中的段交换到磁盘上，因此就是共享内存段机制同样也包括了对磁盘的读写操作。

(2) 连接和无连接

文件和共享内存段就像公告牌一样。数据产生者将信息贴在公告牌上，多个消费者可以同时从公告牌上阅读信息。FIFO 要求建立连接，因为在内核转换数据之前，读者和写者都必须等待着 FIFO 被打开，并且也只有一个客户可以阅读此消息。流 socket 是面向连接的，而数据报 socket 则不是。在某些应用程序中，这些区别起着关键性的作用。

(3) 范围

你希望程序中的消息能传送多远的距离呢？共享内存和命名管道只允许本机上的进程之间通信。通过文件进行传输可以允许不同机器上的进程进行通信。使用 IP 地址的 socket 可以与不同机器上的进程进行通信，而使用 Unix 地址的 socket 却不能。这样，使用哪一种方法进行通信就取决于通信实体间的距离了。

(4) 访问限制

你是希望所有人都能与服务器通信还是只有特定权限的用户才行？文件、FIFO、共享内存以及 Unix 地址 socket 都提供标准的 Unix 文件系统权限。而 Internet socket 则不行。

(5) 竞态条件

使用共享内存和共享文件要比使用管道和 socket 麻烦。管道和 socket 是由内核来管理的队列。写者将数据放进一端，而读者则从另一端将数据读出，进程并不需要考虑其内部结构。

然而对于共享文件和共享内存的访问却不是由内核进行管理的。如果某进程在读文件的过程中，另一个进程正在对文件进行重写，读进程读到的很可能就是不完整或不一致的数据。下一节将会介绍文件锁和信号量的应用。

15.4　进程之间的分工合作

如何处理这些令人恼火的竞态条件呢？客户和服务器若通过共享文件或内存的方式来进行通信，又如何来保证它们正常运行而不出现冲突呢？它们如何分工合作？本节将介绍进程在访问共享资源时所使用的技术：文件锁和信号量。

15.4.1　文件锁

1. 两种类型的锁

考虑两种类型的问题。首先，当客户试图读取文件时，服务器正在重写文件，结果会如何呢？客户读出来的可能就是不完整的数据。上面所提到的日期/时间服务器不太可能遇到这个问题，因为其消息比较短，重写时间较少。但如果是天气预报服务器，其交互的消息比较长，就有可能遇到竞态问题。因此，当服务器在重写文件的时候，客户必须等待服务器完成之后才能开始读。

再考虑一下正好相反的情况。当客户正在一行一行读数据的时候，服务器突然把文件抢过来，将内容删除，然后开始重写数据。客户端看着文件从自己眼皮底下被抢过去而无能

为力。因此，当客户在读取文件的时候，服务器也必须等待客户完成。其他的客户不必去等，因为多个进程一起读文件不会带来任何风险。

为了避免这些问题，需要两种类型的锁。第一种类型为写数据锁，它告诉其他进程："我在写文件，在完成之前任何人都必须等待。"第二种类型的锁为读数据锁，它告诉其他进程："我在读文件，要写文件必须等我完成，要读文件的不受影响。"

2. 使用文件锁进行编程

Unix 提供了 3 种方法锁住打开的文件：flock、lockf 和 fcntl。三者中最灵活和移植性最好的应该是 fcntl。

下面使用 fcntl 锁文件。

(1) 如何给已经打开的文件加读数据锁?

使用 fcntl(fd,F_SETLKW,&lockinfo)

第一个参数是该文件对应的文件描述符。第二个参数 F_SETLKW 说明若必要的话，可以等待其他的进程释放锁。第三个参数指向一个 struct flock 类型的变量。下列代码为一个文件描述符设置读数据锁：

```
set_read_lock(int fd)
{
    struct flock lockinfo;
    lockinfo.l_type = F_RDLCK;                /* a read lock on a region */
    lockinfo.l_pid = getpid();                /* for ME */
    lockinfo.l_start = 0;                     /* starting 0 bytes from.. */
    lockinfo.l_whence = SEEK_SET;             /* start of file */
    lockinfo.l_len = 0;                       /* extending until EOF */
    fcntl(fd,F_SETLKW,&lockinfo);
}
```

(2) 如何在打开的文件上加写数据锁?

使用 fcntl(fd,F_SETLKW,&lockinfo)，并将 lockinfo.l_type 置 F_WRLCK。

(3) 怎样解锁?

使用 fcntl(fd,F_SETLKW,&lockinfo)，并将 lockinfo.l_type 置 F_UNLCK。

(4) 如何只锁住文件的一部分?

使用 fcntl(fd,F_SETLKW,&lockinfo)，并将 lockinfo.l_start 置为开始位置的偏移量，同时将 lockinfo.l_len 置为区域的长度。

3. 基于文件的时间服务器代码

```
/* file_ts.c - read the current date/time from a file
 *   usage: file_ts filename
 *   action: writes the current time/date to filename
 *   note: uses fcntl() - based locking
 */

#include <stdio.h>
```

```
#include <sys/file.h>
#include <fcntl.h>
#include <time.h>

#define oops(m,x) { perror(m); exit(x); }

main(int ac, char *av[])
{
    int fd;
    time_t now;
    char *message;

    if ( ac != 2 ){
        fprintf(stderr,"usage: file_ts filename\n");
        exit(1);
    }
    if ( (fd = open(av[1],O_CREAT|O_TRUNC|O_WRONLY,0644)) == -1 )
        oops(av[1],2);

    while(1)
    {
        time(&now);
        message = ctime(&now);              /* compute time */

        lock_operation(fd, F_WRLCK);        /* lock for writing */

        if ( lseek(fd, 0L, SEEK_SET) == -1 )
            oops("lseek",3);
        if ( write(fd, message, strlen(message)) == -1 )
            oops("write", 4);

        lock_operation(fd, F_UNLCK);        /* unlock file */
        sleep(1);                           /* wait for new time */
    }
}

lock_operation(int fd, int op)
{
    struct flock lock;

    lock.l_whence = SEEK_SET;
    lock.l_start = lock.l_len = 0;
    lock.l_pid = getpid();
    lock.l_type = op;

    if ( fcntl(fd, F_SETLKW, &lock) == -1 )
        oops("lock operation", 6);
}
```

4. 基于文件的时间服务客户端代码

```
/* file_tc.c - read the current date/time from a file
 *     usage: file_tc filename
 *     uses: fcntl() - based locking
 */

#include   <stdio.h>
#include   <sys/file.h>
#include   <fcntl.h>

#define oops(m,x) { perror(m); exit(x); }
#define BUFLEN 10

main(int ac, char *av[])
{
   int fd, nread;
   char buf[BUFLEN];

   if ( ac != 2 ){
      fprintf(stderr,"usage: file_tc filename\n");
      exit(1);
   }

   if ( (fd= open(av[1],O_RDONLY)) == -1 )
      oops(av[1],3);

   lock_operation(fd, F_RDLCK);

   while( (nread = read(fd, buf, BUFLEN)) > 0 )
      write(1, buf, nread );

   lock_operation(fd, F_UNLCK);

   close(fd);
}

lock_operation(int fd, int op)
{
   struct flock lock;

   lock.l_whence = SEEK_SET;
   lock.l_start = lock.l_len = 0;
   lock.l_pid = getpid();
   lock.l_type = op;

   if ( fcntl(fd, F_SETLKW, &lock) == -1 )
      oops("lock operation", 6);
}
```

5. 文件锁：小结

使用 F_SETLKW 参数调用 fcntl 可以使进程挂起直到内核允许进程设置指定的锁。在读取数据之前，客户必须设置读取数据的锁。若服务器对文件加写数据锁，客户只好等待服务器完成。服务器在重写数据之前，也必须对文件加写数据锁，如果这时客户加了一个读数据的锁，那服务器会被挂起直到所有客户释放这个锁。

6. 重要细节：进程可以忽略锁机制

在前面对于文件锁的讨论中，不管客户还是服务器在读或修改文件的时候，程序都是自觉有序地等待，设置及释放文件锁。那么当别的进程设置了锁的时候，其他进程是否可以忽略它，仍旧继续原来的读取或是修改操作吗？答案是肯定的。Unix 的锁机制允许进程通过这种方式合作，但并不强迫它们一定要用。

15.4.2 信号量(Semaphores)

在基于共享内存段技术的时间/日期系统中，共享内存段的作用与基于文件系统的时间/日期系统中的文件是相同的。前面介绍了用锁的机制来解决访问文件的冲突，共享内存段如何来避免数据冲突呢？在共享内存段中是否也存在着读数据锁和写数据锁的概念？不是，但进程使用一个更加灵活的机制来合作：信号量。

信号量是一个内核变量，它可以被系统中的任何进程所访问。进程间可以使用这个变量来协调对于共享内存和其他资源的访问。上一章讨论了如何在特定的情况发生时使用条件变量来通知其他线程。条件对象是进程中的全局变量，而信号量则是系统中的全局变量。

在时间/日期服务器和客户端程序中，如何来使用信号量呢？

1. 计数器及其操作

在无客户读取的时候，服务器将数据写入共享内存段中。同样地，在服务器没有对共享内存段进行写操作的时候，客户可以读取数据。可以将这些规则转换为关于变量值的表达式：

- 客户端等待直到 number_of_writers == 0
- 服务器等待直到 number_of_readers == 0

信号量是系统级的全局变量，这里可以使用两个信号量分别代表读者数和写者数。管理者写变量需要两个操作。

举例来说，读者必须等待写者数为零的时候，才可以将读者数加 1。当某读者读完数据，读者数必须被减一。

同样地，写者也必须等待读者数为零的时候，才可以将写者数加 1。等待读者数为零以及将写者数加 1 是两个独立的操作，必须分开执行，即这两个操作都是原子操作。通过使用信号量来通信的进程可以使用若干个这样的变量，并且独立地进行这些原子操作。

这就是信号量机制的工作原理。进程可以同时处理一组信号量上的多个操作。

2. 一组信号量、多个活动

时间服务器系统使用两个信号量，如图 15.8 所示，并且读者和写者需同时对两个活动集进行操作。

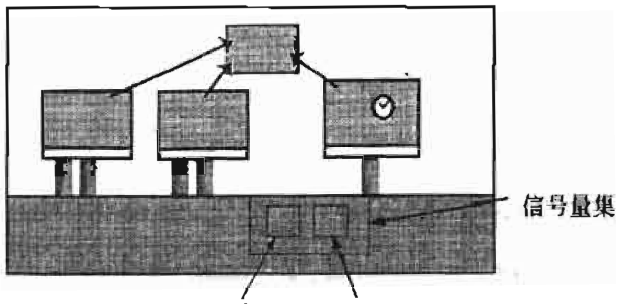


图 15.8　信号量设置：num_readers、num_writers

在修改共享内存之前，服务器必须先对这组活动集进行操作：

- [0] 等候 num_readers 变成 0
- [1] 将 num_writers 加 1

当服务器完成写操作之后，它必须再对下面这组活动集进行操作：

- [0] 将 num_writers 减 1

在客户读取共享内存之前，必须对下面这组活动集进行操作：

- [0] 等待 num_writers 变成 0
- [1] 将 num_readers 加 1

当客户完成任务之后，需要对下面这组活动集进行操作：

- [0] 将 num_readers 减 1

3. 服务器版本：shm_ts2.c

给原来的程序 shm_ts.c 添加信号量得到 shm_ts2.c：

```
/* shm_ts2.c - time server shared mem ver2 : use semaphores for locking
 * program uses shared memory with key 99
 * program uses semaphore set with key 9900
 */

#include  <stdio.h>
#include  <sys/shm.h>
#include  <time.h>
#include  <sys/types.h>
#include  <sys/sem.h>
#include  <signal.h>

#define   TIME_MEM_KEY 99              /* like a filename */
#define   TIME_SEM_KEY 9900
#define   SEG_SIZE ((size_t)100)       /* size of segment */
#define   oops(m,x) { perror(m); exit(x); }

union semun { int val ; struct semid_ds * buf ; ushort * array; };
int      seg_id, semset_id;            /* global for cleanup() */
void     cleanup(int);
```

```
main()
{
    char      *mem_ptr, *ctime();
    time_t    now;
    int       n;

    /* create a shared memory segment */

    seg_id = shmget( TIME_MEM_KEY, SEG_SIZE, IPC_CREAT|0777 );
    if ( seg_id == -1 )
        oops("shmget", 1);

    /* attach to it and get a pointer to where it attaches */

    mem_ptr = shmat( seg_id, NULL, 0 );
    if ( mem_ptr == ( void * ) -1 )
        oops("shmat", 2);

/* create a semset: key 9900, 2 semaphores, and mode rw-rw-rw */

    semset_id = semget( TIME_SEM_KEY, 2,
                        (0666|IPC_CREAT|IPC_EXCL) );
    if ( semset_id == -1 )
        oops("semget", 3);

    set_sem_value( semset_id, 0, 0);            /* set counters */
    set_sem_value( semset_id, 1, 0);            /* both to zero */

    signal(SIGINT, cleanup);

    /* run for a minute */
    for (n=0; n<60; n++ ){
        time( &now );                           /* get the time */
            printf("\tshm ts2 waiting for lock\n");
        wait_and_lock(semset_id);               /* lock memory */
            printf("\tshm_ts2 updating memory\n");
        strcpy(mem_ptr, ctime(&now));           /* write to mem */
            sleep(5);
        release_lock(semset_id);                /* unlock */
            printf("\tshm_ts2 released lock\n");
        sleep(1);                               /* wait a sec */
    }

    cleanup(0);
}

void cleanup(int n)
{
    shmctl( seg_id, IPC_RMID, NULL );           /* rm shrd mem */
```

```
    semctl( semset_id, 0, IPC_RMID, NULL);                  /* rm sem set */
}

/*
 * initialize a semaphore
 */
set_sem_value(int semset_id, int semnum, int val)
{
    union semun initval;

    initval.val = val;
    if ( semctl(semset_id, semnum, SETVAL, initval) == -1 )
        oops("semctl", 4);
}

/*
 * build and execute a 2-element action set:
 * wait for 0 on n_readers AND increment n_writers
 */
wait_and_lock( int semset_id )
{
    struct sembuf actions[2];                   /* action set */
    actions[0].sem_num = 0;                     /* sem[0] is n_readers */
    actions[0].sem_flg = SEM_UNDO;              /* auto cleanup */
    actions[0].sem_op = 0 ;                     /* wait til no readers */

    actions[1].sem_num = 1;                     /* sem[1] is n_writers */
    actions[1].sem_flg = SEM_UNDO;              /* auto cleanup */
    actions[1].sem_op = +1 ;                    /* incr num writers */

    if ( semop( semset_id, actions, 2) == -1 )
        oops("semop: locking", 10);
}

/*
 * build and execute a 1-element action set:
 * decrement num_writers
 */
release_lock( int semset_id )
{
    struct sembuf actions[1];                   /* action set */

    actions[0].sem_num = 1;                     /* sem[0] is n_writerS */
    actions[0].sem_flg = SEM_UNDO;              /* auto cleanup */
    actions[0].sem_op = -1 ;                    /* decr writer count */

    if ( semop( semset_id, actions, 1) == -1 )
        oops("semop: unlocking", 10);
```

}

此程序中，使用信号量集的服务器必须完成下面的 5 个步骤。

(1) 创建信号量集

semset_id = semget(key_t key, int numsems, int flags)

semget 函数创建了一个包含 numsems 个信号量的集合。shm_ts2 程序创建了包含两个信号量的集合。此集合所拥有的权限模式是 0666。函数 semget 返回此信号量集的 ID。

(2) 将所有的信号量置 0

semctl(int semset_id, int semnum, int cmd, union semun arg)

这里使用 semctl 来对信号量集进行控制。此函数的第一个参数是此集合的 ID，第二个参数是集合中某特定信号量的号码，第三个参数是控制命令。如果此控制命令需要参数，那么使用第四个参数向其提供所需的参数。在 shm_ts2 中，使用 SETVAL 命令来给每一个信号量赋一个初值零。

(3) 等待所有读者完成任务之后，服务器将 num_writers 加 1

semop(int semid, struct sembuf * actions, size_t numactions)

函数 semop 对信号量集完成一组操作。第一个参数用来指定信号量集。第二个参数是一组活动的数组。最后一个参数则是该数组的大小。集合中的每一个活动都是一个结构体，它的作用就是“使用选项 sem_flg 来完成对号码为 sem_num 的信号量的操作 sem_op”。整个活动集合被作为组来完成，这一点是关键。上面程序中的函数 wait_and_lock 完成两个操作：等待读者数到零，然后将写者数加 1。这里建了一个包含这两个活动的数组。活动 0 所要完成的事情就是“等待信号量 0 变成 0”。而活动 1 要完成的功能则是“将信号量 1 加 1”。进程挂起直到这两个活动都被完成。只要读者计数器一变成 0，写者计数器立即加 1，然后 semop 函数返回。

使用 SEM_UNDO 标志允许内核在进程退出的时候恢复这些操作。在上面这个程序中，当写者计数器被加 1 的时候，共享内存段是被锁住的。如果在对此计数器做减 1 操作之前，进程非法终止，其他进程则永远无法读取共享内存段的内容了。

(4) 对 num_writers 减 1

在 release_lock 函数中，只需完成一件事情：对写者数减 1。这里使用一个只包含该活动的数组作为参数来调用 semop 函数，从而完成对写者数目的修改。如果这时某客户正在等待，它立即就可以继续执行读操作了。

(5) 删除信号量

semctl(semset_id, 0, IPC_RMID, 0)

任务完成之后，服务器再次调用 semctl 函数，不过这次的目的是删除信号量。

4. 客户端程序：shm_tc2.c

客户端的设计相对容易得多。在程序 shm_tc 中，既不初始化信号量也不删除它们。

```
/* shm_tc2.c - time client shared mem ver2 : use semaphores for locking
 * program uses shared memory with key 99
```

```
 * program uses semaphore set with key 9900
 */

#include  <stdio.h>
#include  <sys/shm.h>
#include  <time.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>

#define  TIME_MEM_KEY 99             /* kind of like a port number */

#define  TIME_SEM_KEY 9900           /* like a filename */
#define  SEG_SIZE ((size_t)100)      /* size of segment */
#define  oops(m,x) { perror(m); exit(x); }
union semun { int val ; struct semid_ds * buf ; ushort * array; };

main()
{
    int   seg_id;
    char * mem_ptr, * ctime();
    long now;

    int   semset_id;             /* id for semaphore set */

    /* create a shared memory segment */

    seg_id = shmget( TIME_MEM_KEY, SEG_SIZE, 0777 );
    if ( seg_id == -1 )
        oops("shmget",1);

    /* attach to it and get a pointer to where it attaches */

    mem_ptr = shmat( seg_id, NULL, 0 );
    if ( mem_ptr == ( void * ) -1 )
        oops("shmat",2);

    /* connect to semaphore set 9900 with 2 semaphores */

    semset_id = semget( TIME_SEM_KEY, 2, 0);
    wait_and_lock( semset_id );

    printf("The time, direct from memory: .. %s", mem_ptr);

    release_lock( semset_id );
    shmdt( mem_ptr ); /* detach, but not needed here */
}

/*
 * build and execute a 2-element action set:
 * wait for 0 on n_writers AND increment n_readers
 */
```

```
wait_and_lock( int semset_id )
{
    union semun sem_info;                      /* some properties */
    struct sembuf actions[2];                  /* action set */
    actions[0].sem_num = 1;                    /* sem[1] is n_writers */
    actions[0].sem_flg = SEM_UNDO;             /* auto cleanup */
    actions[0].sem_op = 0 ;                    /* wait for 0 */
    actions[1].sem_num = 0;                    /* sem[0] is n_readers */
    actions[1].sem_flg = SEM_UNDO;             /* auto cleanup */
    actions[1].sem_op = +1 ;                   /* incr n_readers */

    if ( semop( semset_id, actions, 2) == -1 )
       oops("semop: locking", 10);
}

/*
 * build and execute a 1-element action set:
 * decrement num_readers
 */
release_lock( int semset_id )
{
    union semun sem_info;                      /* some properties */
    struct sembuf actions[1];                  /* action set */
    actions[0].sem_num = 0;                    /* sem[0] is n_readers */
    actions[0].sem_flg = SEM_UNDO;             /* auto cleanup */
    actions[0].sem_op = -1 ;                   /* decr reader count */

    if ( semop( semset_id, actions, 1) == -1 )
       oops("semop: unlocking", 10);
}
```

编译并对程序做如下的测试：

```
$ cc shm_ts2.c -o shmserv
$ cc shm_tc2.c -o shmclnt
$ ./shmserv&
[1] 15533
        shm_ts2 waiting for lock
        shm_ts2 updating memory
$       shm_ts2 released lock
$       shm_ts2 waiting for lock
        shm_ts2 updating memory
$ ./shmclnt
      shm_ts2 released lock
The time, direct from memory: ..Sat Oct 27 17:36:34 2001
$       shm_ts2 waiting for lock
        shm_ts2 updating memory
```

```
  $ ./shmclnt
          shm_ts2 released lock
  The time, direct from memory: ..Sat Oct 27 17:36:40 2001
  $           shm_ts2 waiting for lock
  ipcs
------ Shared Memory Segments --------
key          shmid      owner    perms   bytes   nattch   status
0x00000063 30670854  bruce    777     100     1
------ Semaphore Arrays --------
key          shmid      owner    perms   nsems   status
0x000026ac 262146    bruce    666     2
------ Message Queues ------
key          msqid      owner    perms   used-bytes messages

$     shm_ts2 released lock
      shm_ts2 waiting for lock
$ kill - INT 15533
$ semop: unlocking: Invalid argument
```

上面对程序的测试展示了客户端如何等待服务器解锁。许多客户可以同时运行，每一客户等待服务器计数器变化到 0，然后对客户计数器加 1。如果三个客户同时从共享内存中读取数据，读者计数器也将是三个。服务器只好等三个客户的读者计数器都变为 0 时，才可以进行数据的写操作。

程序中并没有对可能出现的所有情况进行处理。具体来说，如何防止两个服务器程序同时运行？在我们的程序中，服务器仅仅等待客户的读者服务器变为 0 而并没有对其他服务器的写者计数器进行判断。

5. 等待某信号量变为正数

客户端等待着服务器的写者信号量变为零，而同时服务器等待着客户端的读者信号量变为零。但在其他的程序中，也许希望等待的是某个信号量变成正数值。举例来说，也许希望等待信号量的值变为 2。如何来写这样的程序呢？

这里使用一个不太直接的方法：让系统内核对信号量做减 2 操作。信号量不允许为负值，因此系统内核将调用挂起直到信号量的值大于或等于 2。信号量一旦达到 2，某进程就对它做减 2 操作，然后把任何其他要对这个信号量减 2 的进程挂起。

这个操作的 sem_op 成员工作方式如下。

- 若 sem_op 是正值，活动：通过 sem_op 函数对信号量减 2。
- 若 sem_op 是零，活动：挂起直到信号量等于 0。
- 若 sem_op 是负值，活动：挂起直到信号量变成正值。

15.4.3 socket 及 FIFO 与共享的存储

本章的前面部分写了四个版本的时间/日期服务器和客户端程序。socket 版本和 FIFO 版本要相对容易些。客户端连接到服务器，服务器发送数据，然后服务器进程挂起。虽然共

享内存和文件的版本看上去很简单，但它们需要锁和信号量机制来保护数据。要知道加入锁和信号量也是相当复杂的一件事。

然而文件和共享内存机制允许多客户端同时从服务器读取数据，允许客户端和服务器在不同的时刻运行，并且当进程崩溃时，允许数据的保持和恢复。

管道和 socket 也包含了锁的机制。管道和 socket 其实也是保存数据的内存段，它将数据从源端复制到目的端。不同的是管道和 socket 中的锁和信号量是由内核，而不是由进程来管理的。

15.5 打印池

在时间/日期程序中，服务器发数据给客户端。然而另外的一些程序却是以截然不同的方式工作：多客户端发数据给服务器，例如打印服务器上的打印池。那么这种类型的程序如何来设计呢？

15.5.1 多个写者、一个读者

如图 15.9 所示，多用户共享一个打印机。如何使用客户端/服务器模型来设计一个共享打印机的程序呢？多个用户可能会在同时发送打印请求，但是打印机在某一时刻只能打印一个文件。打印程序就必须接收多个并发的输入，并将单个的输出流送到打印设备上。如何来写这个服务器程序呢？它们之间又如何通信呢？

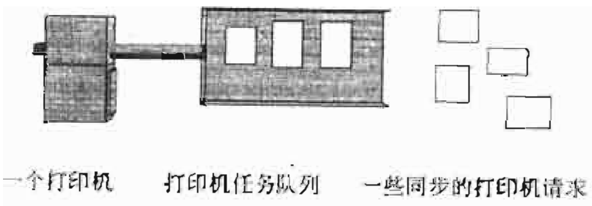


图 15.9　多个数据源、一个打印机

这个程序由哪些功能单元组成？这些单元之间又传递哪些数据和消息呢？

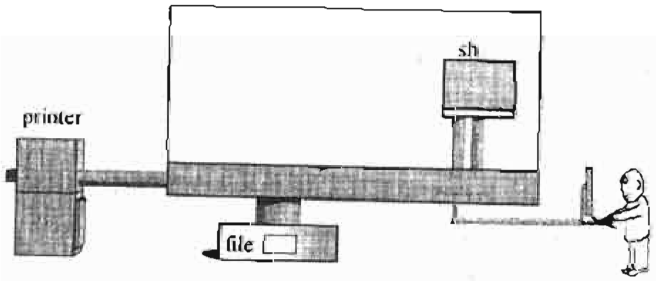


图 15.10　将一个文件传给打印机

在 Unix 系统中打印文件的最简单方法就是使用如下命令：

```
cat filename > /dev/lp1 或者 cp filename /dev/lp1
```

这里/dev/lp1 是打印机设备文件的名称。当然系统中打印设备文件名称并不一定和上面的一样，但在 Unix 系统中将数据传给打印机或其他设备的惟一方法就是通过 open 打开文件，然后使用 write 系统调用将数据写至打印文件中。

可以使用写数据锁吗？

大家已经学习过写数据锁和信号量机制了。为什么不可以自己写一个 cat 或 cp 的打印程序，让它通过写数据锁来防止对设备文件的同步访问冲突呢？

基于锁机制的文件复制程序确实没有问题。考虑一下对打印机加锁后，结果会怎样。若某程序对打印机加锁，其他的文件复制程序都必须挂起等待第一个程序完成任务并释放锁。那么下一步哪个程序执行呢？内核将所有挂起进程中的一个唤醒，但是这个进程却不一定就是排在第二的。显然这样的决定有失公平。允许用户通过复制数据到打印设备文件来实现打印还有另一个问题：若有些人试图作假，他们可以不使用这样一个加锁的程序来打印。第三个问题则是某些文件需要特殊的处理。例如图像文件有可能需要被转换为打印机可以看得懂的图像命令。很多用户并不知道如何将数据转换成通用的格式，那么他们就得不到正确的结果。然而这所有的问题都可以由集中化(客户/服务器模式)来解决。

15.5.2　客户/服务器模型

程序的客户/服务器模型解决了前面提到过的打印的问题。只有一种称为线性打印精灵(line printer daemon)的服务器程序有权限去写数据到打印设备文件中，而其他的用户进程则不行(如图 15.11 所示)。当用户需要打印文件的时候，他们运行一个称为 lpr 的客户端程序。lpr 对文件做了一个复制，然后将复制的文件放在打印任务队列中。用户可以删除或编辑这个文件。并且打印精灵程序可以将图片和格式做转换以使得它们能够正确地被打印出来。

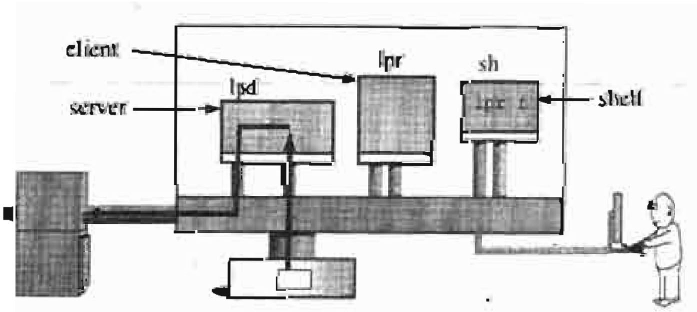


图 15.11　客户/服务器模型的打印系统

客户端和服务器如何通信呢？它们交互哪些数据？客户端将整个文件传给服务器还是客户端仅仅将文件名传给服务器呢？如果服务器和客户不是在同一台机器上，情况又将如何？这是否影响到对通信方式的选择呢？不同版本的 Unix 中有不同的打印系统，有些使用 socket，有些使用命名管道，而另外一些仅使用 fork 和文件。

是否使用集中化的客户/服务器模式就可以不使用锁机制来避免冲突了？可以把系统设计成一个通过构件进行通信和合作的模型来打印一台机器上的文件，也可以用来打印 Internet 上的文件。可以将你自己的思路和不同版本的 Unix 打印系统的设计思路做一个

比较。

15.6 纵观 IPC

本章已经介绍过各种形式的进程间通信方法。下面是一个小结。

方　法	类型	是否可以使用在不同机器上	不同的进程			不同的线程
			P/C	Sib	Unrel	
exec/wait	M		*			
environ	M		*			
pipe	S		*	*		*
kill-signal	M		*	*	*	*
inet socket	S	*	?	?	?	?
inet socket	M	*	?	?	?	?
Unix socket	S		?	?	*	?
Unix socket	M		?	?	*	?
named pipe	S		?	?	*	?
shared mem	R		*	*	*	?
msg queue	M		*	*	*	*
files	R	N	*	*	*	?
variables	M					*
file locks	C	N	*	*	*	*
semaphores	C		*	*	*	?
mutexes	C					*
link	C		*	*	*	?

内容说明：

P/C——父/子关系

Sib——兄弟关系

Unrel——无关进程

M——发送消息

S——使用读和写的数据流

R——随机读取数据

C——用来使任务同步或合作

*——适当的应用

?——不适当的应用

N——适合应用在网络文件系统上

上面的这张表并不包含贝尔实验室的网络工具 TLI 以及它的后续产品。

解释如下：

- fork－execv－argv、exit－wait

用于使用一组参数来调用某个程序，被调用函数将一个整型值返回给其调用者。父进程通过使用 fork 来创建一个新的进程。在此新进程中的程序可以通过调用 execv 来运行新的程序，并传递给新程序一组参数。子进程通过使用 exit 传回一个返回值，同时父进程使用 wait 来接收这个返回值。

这一组调用是面向消息的，它们仅仅可以使用在相关的进程中，且只能在单机上使用。

• environ

系统调用 exec 通过一个叫做 environ 的系统全局变量自动将一组字符串复制进新的程序中。此方法允许进程传值给子进程。由于整个环境被复制给子进程，子进程无法改变父进程的运行环境。

此方法也是面向对象的、单向的，仅仅可以使用在相关的进程中，且只能在单机上使用。

• pipe

管道是由进程创建的单向数据流。它包含连接到内核上的文件描述符。写进一个文件描述符的数据可以从另一个文件描述符读出来。如果进程在创建管道之后调用了 fork，那么新的进程就可以通过同样的管道读写数据。

此方法是面向流的，通常为单向传输，也仅仅可以使用在相关的进程中，且只能在单机上使用。

• kill－signal

信号(signal)是一条从一个进程发往另一个进程的整型消息(使用 kill 系统调用)。接收进程可以通过使用 signal 系统调用来安排一个处理者函数，此函数在信号到来的时候被调用。

此方法是面向消息的，某一时刻单向的，进程必须拥有相同的用户 ID，且只能在单机上使用。

• Internet sockets

Internet sockets 是这样一条链接，它的两个端点是通过特定的端口号建立起来的。字节流通过 socket 进行传输，从一个进程到达另一个进程，类似于某人在波士顿打电话给在东京的朋友。Internet sockets 有两种主要的实现方式：流 socket 和数据报 socket。这两者皆可以双向传输。流 socket 更类似于文件描述符：程序员使用 write 和 read 调用来发送和接收数据。数据报 socket 则类似于明信片：写者将缓存中的一块数据发给读者。所有的交互都是以数据缓存的形式完成，而不是字节流。

有面向消息和面向流两个版本，双向传输，可以在无关进程中使用，可以通过网络传输数据。

• Named Sockets

命名 socket，又称 Unix 域 socket。它使用文件名作为地址而不是主机名一端口号对。命名 socket 同时支持流和数据报版本。因为这种方式使用文件名而不是主机一端口作为地址，所以它们仅仅可以连接同一机器上的进程。

这种方法有面向消息和面向流两个版本，双向传输，可以在无关进程中使用，只能工作在单机上。

• Named Pipes(FIFOs)

命名管道的工作方式类似于一个常规管道，但是它可以连接两个无关的进程。命名管

道由文件名来标志。写者使用 open 调用来打开文件并写数据，读者同样使用 open 调用打开文件读数据。这种方法比命名 socket 使用起来方便很多，但是它只可以单向传输。

此方法是单向传输的、面向流的，可以连接无关进程，只能工作在单机上。

- File Locks

Unix 允许进程对文件的访问设置锁。进程可以对文件的某一段加锁，使自己可以单独地对这一段进行改动。另一个试图锁住此文件的进程将被挂起，直到这个文件被解锁。文件锁机制允许进程间进行通信，让所有的进程都知道是哪一个进程正在读取或修改文件。这里可以使用系统调用 flock、lockf 和 fcntl 来设置或测试文件锁。在某些系统中，这些锁是被强制加上的。

这种方法面向消息，多个无关进程间可以同时交互，但只能在单机上工作。

- Shared Memory

每个进程都有其自己的数据空间。程序所定义的任何变量或在运行时刻分配的空间都只有对该进程是可见的。进程可以通过使用 shmget 和 shmat 调用来创建可以被多个进程共享的内存段。由一个进程写入共享内存段的数据可以被别的对此内存段有访问权限的进程读出。这是 IPC 中最为有效的一个方法，因为所有的通信并不需要数据的传输。

此方法面向随机访问，多个无关进程间可以同时交互，但只能在单机上工作。

- Semaphores

信号量是系统级的变量，程序之间可以通过信号量来进行通信。进程可以对信号量做增 1 操作、减 1 操作或等待信号量到某个特定的值。信号量类似于许可证服务器上的许可证。当进程需要使用资源的时候，它对信号量做减 1 操作（取一个许可证）。如果此时许可证已经用完了，进程挂起直到其他进程对信号量做了增 1 操作。信号量应用在各种各样的程序中。

此方法是面向消息的，多个无关进程间可以同时交互，但只能在单机上工作。

- Message Queues

消息队列的工作原理类似于 FIFO，但它并不是以文件名来标志。进程可以将消息加到队列中，然后由其他进程将数据从队列中取出。多个队列可以被多个进程所共享。

这种方法是面向消息的、单向传输的，且只能工作在单机上。

- Files

文件可以被多个进程在同一时刻打开。如果某进程将数据写入一个文件，另外的进程可以从该文件中读出数据。若大家都使用一个经过巧妙设计的交互协议，很多复杂的通信都可以通过古老的文件机制来实现。

此方法面向随机访问，多个无关进程间可以同时交互，网络文件系统（NFS）可以支持跨机器的多进程通信。

15.7 连接与游戏

本章介绍了很多进程之间传递数据的方法。Unix 的系统内核管理着进程、文件和设备，并且对管道、socket、文件、共享内存以及信号进行操作使它们可以传输数据。对于某些程序

来说，创建和管理连接与数据的传输是最主要的部分。

Unix 的开发者之一 Ken Thompson，在 1978 年写道：

"与其说 Unix 的内核是一个完整的操作系统，还不如说它是个 I/O 多路复用器(multiplexer)。从这种观点出发，许多其他操作系统中的特征在 Unix 内核中是找不到的……这许多的功能都是由用户程序使用内核调用来实现的。[①]"

在第 1 章中，讨论了命令 bc、一个 Web 服务器还有一个网络桥牌游戏。接着写了一个 bc 程序和两个 Web 服务器程序。那么如何写网络桥牌游戏呢？可以使用屏幕控制程序来作为用户界面，使用 socket 来连接两端。那么哪一端是服务器？哪一端是客户呢？如何使用锁机制？你所需要的所有的技术都包含在本书的章节中。在 Unix 上进行编程并没有想象中的那么难，可是也并非是一件容易的事。

说到游戏和网络，让我们回忆一下 Dennis Ritchie 是如何来描述用来引出 Unix 的空间探险游戏的：

"最开始写在 Multics 系统上……，这并不亚于对太阳系诸星体运动的模拟。你还必须让玩家驾驶着太空船四处巡视，观看空间的景色并且可能在行星或其卫星上登录。"

驾驶着太空船四处巡视，观看空间的景色并且可能在行星或其卫星上登录不就像生活中的网络冲浪一样吗？可能冲浪并不是最好的比喻。但是确实人们打开他们的浏览器，到全世界四处浏览，Web 服务器将各处的景象返回。人们使用 telnet、ssh 还有 ftp 登录到其他机器上。也许 Internet 恰巧就是 Ritchie 和 Thompson 在 1969 年开始模拟的广阔空间的实现吧！

小　结

1. 主要内容

- 许多程序都包含一个或多个进程，进程间通过共享数据或传递数据进行通信。举例来说，两个人通过使用 Unix 的 talk 命令进行对话，他们就运行了两个进程，将数据从键盘和 socket 传输到屏幕和 socket。
- 某些进程需要从多个源端接收数据，并将数据送到多个目的地。select 和 poll 调用允许进程等待多个文件描述符的输入。
- Unix 提供了许多方法来进行数据在进程间传输。命名管道和共享内存是同一机器上的进程间通信使用的两种技术。通信方法的区别在于它们的速度、所传输的消息类型、所需的范围、限制访问权限的能力以及防止数据冲突的能力。
- 文件锁是进程间使用的避免对文件访问冲突的技术。
- 信号量是进程合作时所使用的系统级的变量。进程挂起等待另一进程改变信号量的值。

2. 下一步做什么

学习 Unix 系统编程的最好的办法就是不断的读程序，写程序。大家可以在网上找到大

① "Unix Implementation" *Bell System Technical Journal*, vol. 56, no. 6, 1978.

量的信息，以及介绍 Unix 内部实现和编程接口的书籍。大家多注意一下每天都使用的程序还有一些吸引你的新程序。通过使用、学习，并且经常自己来实现一些已经存在的程序，就可以更深、更精、更广泛地了解 Unix 的编程了。

3. 习题

15.1　在 talk 程序中，为何不用线程从文件描述符中读取数据呢？一个线程可以从键盘读取数据，而另一线程从 socket 读取数据。若程序由多线程方案来实现，会出现哪些新问题呢？

15.2　talk 程序在大多数的时候都是在读写单个字符，但在传输数据的时候使用的是数据流。使用数据报 socket 的优缺点各是什么？

15.3　基于 FIFO 的时间服务器在执行到 date > /tmp/time_fifo 的时候挂起直到某一客户端打开 FIFO 来读取数据。若服务器挂起的时间很长，客户端是在服务器挂起时能够接收到时间还是在服务器被唤醒的时候能接收到时间？为什么？

15.4　看一下系统调用 mmap。mmap 将文件的某一段模拟成内存中的一个数组，从而允许程序不使用 lseek 就可以对文件进行随机的访问。使用 mmap 与使用文件或共享内存的方式来实现进程间通信有何区别？和别的方法相比，使用 mmap 又有何优缺点？

15.5　talk 中包含了两个相连的进程。使用一下 talk，看一看连接是如何建立的？其中又包含了哪些程序呢？

4. 编程练习

15.6　参考 select 和 poll 调用的使用手册，看看你的系统中是否都支持。在有些系统中，其中一个是真的系统调用，而另外一个则是由那个真的系统调用模拟出来的。使用 poll 来重写程序 selectdemo.c。

15.7　编写使用下列方法的时间/日期服务器和客户端程序。

(1) 使用 IP 地址的数据报 socket。

(2) 使用 Unix 域地址的流 socket。

15.8　编写基于 FIFO 的时间/日期服务器和客户端程序。

15.9　多个共享内存的服务器：

(1) 可以在同一时刻运行两个共享内存的服务器吗？为什么？做一下试验。

(2) 修改服务器程序中的 wait_and_lock 函数，使服务器可以挂起等待直到运行的服务器数目变为 0。

15.10　在使用文件的版本中，用文件锁来保护对共享文件的访问。重写此程序，用信号量来代替文件锁。

15.11　在使用共享内存的版本中，用信号量来保护对共享内存的访问。重写此程序，

用文件锁来替代信号量。当然这时需要一个共享的文件。

15.12　若用户太多,共享内存的信号量解决方案就无法报告正确的结果。考虑一下这样的模式:读者 A 将读者计数器增至 1。接下来,读者 B 将读者计数器增至 2。这时,读者 A 已经读数据完毕,将读者计数器减 1,但读者 C 又将计数器增 1。因此读者计数器这个时候仍为 2。之后,读者 B 结束,A 重来,C 结束,然后 B 又重来……。因此无论什么时候,共享内存都在被读取。解释一下为什么这种情况阻止了写者更新时间。修改此系统,使得写者可以防止新的读者锁住共享内存段。

15.13　编写一个 cp 命令的新版本打印机程序。它使用写数据锁来防止对输出文件的同步访问冲突。在你的机器上使用这个程序同时打印两个文件:

```
printcp file1 /dev/lp1 & printcp file2 /dev/lp1&
```